

А.Фёдоров

**Horland Pascal:
активное использование
turbo Vision 2.0**



А. Фёдоров

Borland Pascal:
практическое использование
Turbo Vision 2.0



Издательство "Диалектика"
Киев 1993

ББК 62.577

Ф37

Ф37

А.Фёдоров. Borland Pascal: практическое использование
Turbo Vision 2.0

Киев: "Диалектика", 1993.-272 с., ил.

ISBN 5-7707-5126-6

Книга предназначена для тех, кто разрабатывает приложения на основе объектно-ориентированной библиотеки Turbo Vision. Основное внимание уделяется описанию новых функциональных возможностей, появившихся в версии 2.0; приводится описание многочисленных расширений, дополнений и примеров их практического использования, методологических особенностей использования Turbo Vision 2.0.

ББК 62.577

Группа подготовки издания:

Ведущий редактор П.В.Гусак

Технический редактор Т.Л.Юрчук

Рассылка по почте А.Н.Тимченко

тел. (044) 266-4074

Учебное издание

А.Фёдоров. Borland Pascal: практическое использование
Turbo Vision 2.0

Подписано к печати 27.08.1993. Формат 84x108 1/32. Бумага офсетная. Гарнитура Таймс. Печать офсетная. Усл.печ.листов 14.5
Тираж 25000 экз. Заказ № 0213/86. Цена договорная.

Подготовлено к печати издательским отделом фирмы "Диалектика".
Украина, 252124 Киев-124, а/я 506. Телефон/факс (044) 266-4074.

ISBN 5-7707-5126-6

© А. Фёдоров

© Издательство "Диалектика", 1993, обработка и оформление

Все названия продуктов являются зарегистрированными торговыми
марками соответствующих фирм.

Отпечатано с компьютерного набора на комбинате печати издательства
"Пресса Украины".

Украина 252047, Киев-47, проспект Победы, 50

ОТ АВТОРА

Данная книга посвящена практическому использованию объектно-ориентированной библиотеки Turbo Vision, разработанной фирмой Borland International. Появившаяся впервые в комплекте с компилятором Turbo Pascal 6.0, эта библиотека завоевала огромную популярность среди отечественных программистов. Последующий перенос Turbo Vision в среду компилятора Turbo C++ сделал ее доступной и для любителей этого языка.

Приступая к написанию этой книги, я планировал ее как сборник различных практических советов по использованию библиотеки Turbo Vision. Книга обобщает мой более чем двух-летний опыт использования и самой библиотеки и объектно-ориентированного расширения языка Pascal. Предполагается, что читатели знакомы с основными принципами организации Turbo Vision и имеют опыт ее практического использования. Книга должна рассматриваться как приложение к стандартной документации и ни в коей мере не заменяет ее. Я настоятельно рекомендую вам приобрести исходные тексты стандартной библиотеки компилятора Turbo Pascal. Этот продукт, называемый Turbo Pascal Run-time Library включает в себя полный исходный текст библиотеки Turbo Vision. Ознакомление с реализацией библиотеки поможет вам не только разобраться в том, как создаются коммерческие объектно-ориентированные библиотеки классов, но и послужит отличным источником знаний.

Эта книга отличается от предыдущих, посвященных данной теме, тем, что в ней вы не найдете ни длинных иерархий объектов, ни подробных описаний отдельных методов или объектов - для этого существует документация или справочная система. Я сосредоточил свое внимание на практической стороне - вы обнаружите большое число примеров использования различных объектов, примеры расширения функциональности объектов и создания новых объектов на базе уже существующих. Также я попытался разъяснить ряд концепций более простым языком. Там, где это необходимо, приводятся некоторые теоретические сведения, но в основном - это набор практических советов. После публикации серии статей по практическому использованию Turbo Vision в журнале "PC Magazine/Russian Edition" я получаю большое число писем, в которых меня просят продолжить эту тему.

Появление новой версии Turbo Vision в составе компилятора Turbo Pascal/Borland Pascal 7.0 еще раз доказало, что заложенные в ней концепции долговечны. Пока будет использоваться среда DOS, Turbo Vision будет жить. Существующие реализации Turbo Vision в графическом режиме еще больше расширяют функциональность этой библиотеки, приближая ее к Microsoft Windows.

О структуре этой книги. Она построена по принципу: от простого - к сложному. Вначале рассматриваются основные концепции, необходимые для понимания функционирования библиотеки в целом. Отдельные главы посвящены группам однородных объектов: рассматриваются способы их применения и примеры расширения. Каждый пример является законченной программой - такой подход кажется мне более правильным, чем, например, построение одной большой программы из частей: в результате вы теряете мысль, а листинг, растянутый на десятки страниц, становится просто бесполезным. В приложение вынесены те материалы, которые, на мой взгляд, могут пригодиться пользователям Turbo Vision, но не оформлены должным образом в документации.

Я буду признателен всем, кто пришлет свои пожелания и замечания по этой книге по адресу:

103062, Москва, Фурманьный пер. д. 2/7 кв. 2

или воспользуются электронной почтой:

alex@computerpress.msk.su

Благодарю вас за интерес к этой публикации.

Благодарности

Я хочу выразить благодарность своей семье, которая с пониманием отнеслась к этому проекту, особенно сыну Алексею, который не сильно отвлекал меня в процессе написания этой книги. Также я благодарен Дмитрию Рогаткину за ценные советы и участие в обсуждениях. Хочу выразить благодарность сотрудникам отдела технического сопровождения фирмы Borland International (Scotts Valley, California) за помощь в сборе материалов для данной книги, а также Зака Урлокера (Zack Urlocker), предоставившего бета-версию компилятора Borland Pascal 7.0.

Автор

Июль 1993 года

ГЛАВА 1. Turbo Vision.

Ключевые понятия

В этой главе рассматриваются ключевые понятия Turbo Vision, знакомство с которыми полезно не только для понимания материалов, приводимых в данной книге, но и является залогом успешного использования этой библиотеки.

Библиотека Turbo Vision вводит нас в мир объектно-ориентированного программирования. Все здесь считается объектом. Сама программа тоже является объектом - приложение строится на базе объекта TApplication, который является наследником более абстрактного объекта TProgram (который описывает свойства некоторой абстрактной программы). Эти объекты более подробно рассматриваются в следующей главе. Здесь же мы заметим, что идеология Turbo Vision существенно отличается от привычной - мы говорим о прикладной программе как об объекте. Такой объект, как, впрочем, и обычная программа, имеет три состояния (метода в случае с объектом TApplication) - инициализация, выполнение и завершение.

Предваряя рассмотрение ключевых понятий Turbo Vision, отметим, что успех использования этой библиотеки заключается в четком понимании идеологии Turbo Vision и следовании основным правилам, которые будут сформулированы в этой главе.

Управление данными или управление событиями?

При построении интерфейсов прикладных программ широко используется метод управления, называемый управлением данными. В этом случае прикладная программа опрашивает клавиатуру и в соответствии с нажатой клавишей, вызывает ту или иную подпрограмму. Обычно в такой программе имеется цикл (он может быть бесконечным), в котором каждому конкретному событию (например, нажатию определенной клавиши) соответствует вызов некоторой подпрограммы. Отмечу, что обработка нажатий клавиш, а также, действий с манипулятором мышь требует дополнительных усилий: если в стандартной библиотеке присутствуют функции, позволяющие

анализировать действия с клавиатурой, то для обработки манипулятора мышью требуется использование дополнительных библиотек.

Библиотека Turbo Vision построена по принципу управления событиями (с этой точки зрения Turbo Vision схожа с Microsoft Windows). Весь ввод обрабатывается Turbo Vision, причем вводимая информация помещается в специальную структуру, называемую событием и передается соответствующему интерфейсному элементу. При таком подходе прикладная программа может не различать источника ввода: например, нажатие кнопки может осуществляться как с клавиатуры, так и "мышью" - реакция будет одна и та же. Реализация такого подхода стандартными средствами может быть довольно комплексной. В Turbo Vision возможны следующие события:

Событие	Источник
evMouseDown	Нажата кнопка мыши
evMouseUp	Отпущена кнопка мыши
evMouseMove	Мышь перемещена
evMouseAuto	Повторяющиеся действия с мышью
evKeyDown	Нажата клавиша на клавиатуре
evCommand	Команда от интерфейсного элемента
evBroadcast	Сообщение от интерфейсного элемента

Для большинства приложений этого набора событий вполне достаточно. В следующем разделе показано, как определить новое событие.

Помимо этого, все интерфейсные элементы имеют ряд предопределенных реакций на некоторые события, например, окна могут максимизироваться, перемещаться и изменять свой размер, меню и строки состояния могут реагировать на нажатие определенных комбинаций клавиш или активацию их элементов мышью и т.д. Подход, предлагаемый Turbo Vision, освобождает разработчика от задачи по созданию средств обработки нажатий клавиш на клавиатуре и действий с манипулятором мышью - они "встроены" в эту библиотеку.

Отмечу, что возможно использование механизма обработки сообщений отдельно от Turbo Vision. Для этого в

прикладной программе необходимо использовать модуль Drivers:

```
////////////////////////////////////
NO_TV.PAS: Использование механизма обработки сообщений
отдельно от Turbo Vision
////////////////////////////////////}

uses Drivers,CRT;
Var
  Event : TEvent;
  Done : Boolean;

Begin
  ClrScr;
  InitEvents;
  HideMouse;
  Done := False;
  Repeat
    GetKeyEvent(Event);
    Case Event.KeyCode of
      kbLeft : GotoXY(WhereX-1, WhereY);
      kbRight : GotoXY(WhereX + 1, WhereY);
      kbUp : GotoXY(WhereX, WhereY-1);
      kbDown : GotoXY(WhereX, WhereY + 1);
      kbAltX : Done := True;
    End;
  Until Done;
End.
```

В приведенном выше примере используются функции модуля Drivers, которые позволяют получать события с клавиатуры и обрабатывать их, хотя сам интерфейс приложения может быть создан средствами другого пакета. Таким же образом можно обрабатывать события от манипулятора мышь: для этого необходимо использовать функцию GetMouseEvent. Конечно, показанный выше способ использования части Turbo Vision нельзя считать правильным, но бывают ситуации, в которых это необходимо.

Определение новых событий

Что делать в том случае, если нам необходимо ввести новый тип события? Для этого необходимо перепоределить метод GetEvent. Пусть новым событием будет событие evKeyUp - клавиша на клавиатуре отпущена. Введем допущение. Пусть имеется функция KeyUp, которая возвращает код отпущенной клавиши (реализацию этой функции оставляем читателям в качестве упражнения). Тогда реализация метода GetEvent будет выглядеть следующим образом:

```

Procedure TDemoApp.GetEvent(Var Event : TEvent);
Var
  Key : Byte;
Begin
  TApplication.GetEvent(Event);
  If Event.What = evNothing Then
    Begin
      Key := KeyUp;
      If K <> 0 Then
        Begin
          Event.What := evKeyUp;
          Event.ScanCode := Key;
        End;
      End;
    End;
  End;
End;

```

Событие *evKeyUp* должно быть задано константой, например:

```

Const
  evKeyUp = $0400;

```

Затем у тех объектов, которые должны обрабатывать данное событие, необходимо установить соответствующий флаг:

```

EventMask := EventMask OR evKeyUp;

```

Переопределение метода *GetEvent* удобно еще и в том случае, когда необходимо записывать события (для из последующего воспроизведения) или при создании макрокоманд. В этом случае в переопределенном методе *GetEvent* вызывается метод *TApplication.GetEvent*, за тем мы проверяем, например, что нажата комбинация клавиш Ctrl-Alt-A и вызываем метод, который выполняет действия, приписанные данному макросу.

Такой поход также удобен для создания демонстрационных версий программ: вместо ввода с клавиатуры или манипуляций с "мышью" весь поток событий берется из файла.

Метод Idle

Необходимо напомнить еще одно понятие, связанное с обработкой событий. Когда в системе ничего не происходит, вызывается метод *TApplication.Idle*. Переопределив этот метод, мы можем предусмотреть в нем обновление содержимого каких-либо специализированных объектов. В примерах, поставляемых в комплекте с Turbo

Vision есть объект *THeapView*, с помощью которого отображается размер свободного пространства в куче. Для обновления содержимого этого объекта в переопределенном методе *Idle* вызывается метод *Update*:

```
Procedure TDemoApp.Idle;  
Begin  
  HeapViewer.Update;  
Inherited Update;  
End;
```

В Turbo Vision метод *Idle* используется для вызова метода *TStatusLine.Update* в результате которого происходит обновление строки состояния.

Функция Message

Эта функция, определенная в модуле *Views* предназначена для отправления сообщения некоторому объекту. Функция реализована следующим образом: все параметры собираются в запись типа *TEvent*, которая затем передается на обработку методу *HandleEvent* объекта, указанного в качестве параметра *Receiver*. Такой объект должен быть отображаемым объектом - наследником объекта *TView*. Функция возвращает результат обработки сообщения, который хранится в поле *Event.InfoPtr*. Если объект обработал сообщение успешно, то поле *Event.InfoPtr* будет содержать указатель на объект, который обработал сообщение. В противном случае, значение этого поля будет *Nil*. Параметр *InfoPtr* может использоваться для передачи данных объекту. Для этого необходимо присвоить ему значение указателя на запись, хранящую передаваемые данные. Если необходимо уведомить получателя об источнике сообщения, полю *InfoPtr* должно быть присвоено значение *@Self*.

Передача сообщения широко используется самими объектами Turbo Vision. Рассмотрим, например, список и вертикальную полосу прокрутки. Когда положение бегунка изменяется, полоса прокрутки посылает сообщение, указывающее на это изменение:

```
Message(TopView, evBroadcast, cmScrollBarChanged, @Self);
```

где *TopView* - это владелец объекта *TScrollBar*.

Отображаемые и неотображаемые объекты

В библиотеке Turbo Vision существует два типа объектов: отображаемые и неотображаемые. Нередко отображаемые объекты называют видимыми, что неправильно, так как такой объект может быть невидим в данное время (скрыт другими объектами или вызван его метод *Hide*). Предком всех отображаемых объектов является объект *TView*, который и задает их основные свойства. Более подробно отображаемые объекты рассматриваются ниже. Заметим, что отличительным свойством данных объектов является наличие у них метода *Draw*, который вызывается каждый раз при необходимости перерисовки объекта.

Отображаемые объекты

Отображаемые объекты являются объектами Turbo Vision, реализующими элементы пользовательского интерфейса. Предком всех отображаемых объектов является объект *TView*. Он и задает основные свойства всех отображаемых объектов в Turbo Vision. Обычно экземпляры объекта *TView* не создаются, а создаются объекты-наследники, которые используют его свойства и расширяют их. Среди свойств, передаваемых объектом *TView* своим наследникам, необходимо выделить следующие: *GrowMode* (поле объекта *TView*) - позволяет задать режим изменения размеров объекта при изменении размеров его владельца (о владельцах мы поговорим чуть позже), *DragMode* - позволяет задать режим перемещения объекта с помощью манипулятора мышь, *Options* - позволяет указать реакцию на события, последовательность их обработки, а также местоположение объекта, *EventMask* - позволяет указать, классы событий, обрабатываемых данным объектом.

Например, в конструкторе объекта *TSomeObject* могут быть заданы следующие значения этих полей:

```
Constructor TSomeObject.Init;  
Begin  
....  
  
GrowMode := gfGrowRel;  
DragMode := dmLimitLoY;  
Options := ofCentered;  
EventMask := evMouseDown + evCommand;
```

....
End;

Свойствами данного отображаемого объекта будут следующие: его размер будет изменяться относительно размера владельца, при перемещении будет проверяться выход за левую границу по оси Y, объект будет находиться в центре своего владельца и будет реагировать на нажатие кнопок на манипуляторе мышь и командные события.

Центральным методом каждого отображаемого объекта является метод *Draw*, в котором происходит непосредственная отрисовка объекта. Метод *Draw* отвечает за отображение всей области, занимаемой объектом, и способен отобразить объект в любое время. Такая схема использования отображаемых объектов отличается от большинства оконных систем, где в большинстве случаев каждый объект сохраняет область экрана, которую он занимает, и восстанавливает ее при необходимости.

Координатная система и объект *TRect*

В *Turbo Vision* для задания координат всех отображаемых объектов используется объект *TRect*: конструктор практически любого объекта-наследника объекта *TView* - имеет параметр *Bounds* типа *TRect*. Ниже приводятся примеры, которые помогут разобраться вам в действии методов этого объекта.

Вызов метода

R.Assign(5,5,75,15);

выполняет следующее присваивание:

R.A.X = 5; R.A.Y = 5; R.B.X = 75; R.B.Y = 15;

Вызов метода

R.Grow(-1,-1);

задает следующие координаты:

R.A.X = 6; R.A.Y = 6; R.B.X = 74; R.B.Y = 14

Очень часто в примерах используются следующие присваивания:

R.A.Y := R.B.Y-1;

В этом случае координаты будут иметь следующие значения:

R.A.X = 5; R.A.Y = 14; R.B.X = 75; R.B.Y = 15;

Перечисленные выше операции используются наиболее часто. Остальные методы объекта *TRect* используются внутри методов других отображаемых объектов.

При задании координат отображаемых объектов необходимо помнить, что для объектов, которые помещаются внутри других объектов, используется координатная область владельца.

Когда в конструкторе *Init* отображаемого объекта мы указываем значение параметра *Bounds*, координаты и размер этого объекта сохраняются следующим образом:

в поле *Origin* заносятся значения:

Origin.X := *Bounds.A.X*

Origin.Y := *Bounds.A.Y*

в поле *Size* заносятся значения:

Size.X := *Bounds.B.X* - *Bounds.A.X*

Size.Y := *Bounds.B.Y* - *Bounds.A.Y*

Для изменения размера и местоположения отображаемого объекта можно использовать методы *TView.ChangeBounds* и *TView.SetBounds*. В качестве параметра для этих методов указывается *Bounds* типа *TRect*.

В Turbo Vision существует два метода, позволяющие определить размер отображаемого объекта. Метод *GetExtent* возвращает размер и местоположение объекта в локальных координатах: при вызове *SomeObject.GetExtent(R)* поля *R.A.X* и *R.A.Y* всегда содержат значение 0,0, а поля *R.B.X* и *R.B.Y* - размер объекта. Для получения координат объекта относительно его владельца необходимо использовать метод *GetBounds*. В этом случае поля *R.A.X* и *R.A.Y* будут содержать значение поля *Origin* данного объекта, а поля *R.B.X* и *R.B.Y* - координаты правого нижнего угла объекта.

Для перемещения отображаемого объекта используется метод *MoveTo*, а для изменения его размеров - метод *GrowTo*. Изменение размера объекта и его перемещение может быть осуществлено за один шаг при помощи метода *Locate*.

Модальные и выбранные объекты

Эти три понятия играют очень важную роль в понимании взаимодействия отображаемых объектов Turbo Vision.

Модальным называется объект, который не позволяет переключиться на другой объект без завершения работы с ним. Например, если отображается модальная панель диалога, вы не сможете переключиться на окно, пока не нажмете одну из кнопок в этой панели или не закроете панель. Модальным также всегда является объект *TApplication*.

Выбранным называется объект, который в данный момент обрабатывает сообщения в первую очередь. Например, выбранные окна имеют специальный тип рамки. Если выбранным является окно редактора, то весь ввод с клавиатуры будет отображаться именно в этом окне.

Объектом в фокусе называется объект (обычно включенный в группу), который в данный момент обрабатывает сообщения. Обычно переключение фокуса в группе осуществляется с помощью клавиши Tab.

Групповые объекты

Объект *TGroup* - наследник объекта *TView*, предназначен для создания групп отображаемых объектов - объектов, которые функционируют совместно. Примером таковых объектов могут служить окна и панели диалога. Обычно группа имеет владельца. Каждый объект, включенный в группу имеет поле *Owner* (типа *PView*), которое содержит указатель на владельца данного объекта. В группе поддерживается список объектов, расположенных в ней, поле *Next* указывает на следующий объект в группе. По мере того как пользователь взаимодействует с программой, состояние группы может изменяться - могут добавляться новые элементы, изменяться значения уже существующих и так далее. Для работы с элементами группы используются специальные методы-итераторы, как например, *ForEach*. Другим примером группы является объект *TApplication*: он "владеет" объектами *TMenuBar*, *TDesktop* и *TStatusLine*.

Включение объекта в группу происходит с помощью метода *Insert*. Например, при создании панели диалога с двумя кнопками происходят следующие действия:

- создается новый объект типа *TDialog*
- в этот объект "включаются" кнопки
- объект *TDialog* становится владельцем двух объектов типа *TButton*

- при выполнении метода *ExecView* отображается сама панель диалога и все включенные в нее объекты

Группа является как бы контейнером, содержащим отображаемые объекты, которые должны работать вместе, например, панель диалога. Метод *HandleEvent* группы просто передает управление методам *HandleEvent* каждого объекта, включенного в группу, метод *Draw* вызывает методы *Draw* всех объектов, то же самое происходит при перемещении или изменении размеров объекта.

Неотображаемые объекты

К этой категории относятся объекты, которые непосредственно не взаимодействуют с экраном. Такие объекты используются отображаемыми объектами Turbo Vision для собственных целей. Неотображаемые объекты реализованы в модуле *Objects*, который может применяться в программах не использующих отображаемые объекты Turbo Vision. Модуль *Objects* является общим для Turbo Vision и Object Windows. Это делает возможным использование неотображаемых объектов как в среде DOS, так и в среде Windows. Более подробно неотображаемые объекты рассматриваются в главе 9.

Разное

В этом разделе собраны некоторые замечания по использованию ряда функций, включенных в Turbo Vision, которые по тем или иным причинам не могут быть включены в другие главы.

Функция CStrLen

Эта функция, определенная в модуле *Drivers*, возвращает размер строки, которая содержит специальные символы (~), указывающие на командную клавишу. Обычно, такие строки используются для описания элементов меню и строк состояния. Функция *CStrLen* возвращает число символов в строке, не включая специальные символы. Например, вызов функции *CStrLen('F~ile')* возвратит 4.

Константа ErrorAttr

Эта константа типа *Byte* со значением *\$CF* определена в модуле *Views*. Она задает атрибут, с помощью которого отображаются объекты, цветовой индекс которых выходит за границу текущей палитры. Атрибут *\$CF* задает мерцающие белые символы на красном фоне. Если какой-либо отображаемый объект в вашей программе имеет такие атрибуты, вам необходимо проверить или переопределить метод *GetPalette* этого объекта.

Процедура FormatStr

Эта процедура, определенная в модуле *Drivers* используется для форматного отображения строк. Обычно эта процедура используется для подстановки параметров в предопределенные строки такие, как сообщения об ошибках. Строка должна содержать как текст, так и символы преобразования формата:

'File %s is %d bytes in size.'

Символы преобразования %s и %d указывают на использование строки и десятичного значения, которые должны быть подставлены вместо них. Значения для подстановки указываются в параметре *Params*. Параметр *Params* содержит данные для каждого элемента в строке. Существует два способа инициализации параметра *Params*:

- используя запись;
- используя массив типа *LongInt*.

В приведенных ниже примерах показано, как используется тот и другой способ. Использование записи является более простым способом, так как каждый параметр является полем записи, тогда как использование массива позволяет изменять число параметров динамически.

```
{//////////}
Использование записи
//////////}
Uses
  Drivers, Objects;

Type
  TParamRec = Record
```

```

FName : PString;
FBytes: LongInt;
End;

Var
ParamRec : TParamRec;
ResultStr: String;

Begin
ParamRec.FName := NewStr('DEMO.TXT');
ParamRec.FBytes := 654321;
FormatStr(ResultStr, 'File %s is %d bytes in size.', ParamRec);
WriteLn(ResultStr);
ReadLn;
End.

{////////////////////
Использование массива
////////////////////}

Uses
Drivers, Objects;

Type
TParamArray = Array[0..1] of LongInt;

Var
ParamArray : TParamArray;
ResultStr : String;

Begin
ParamArray[0] := LongInt(NewStr('DEMO.TXT'));
ParamArray[1] := 654321;
FormatStr(ResultStr, 'File %s is %d bytes in size.', ParamArray);
WriteLn(ResultStr);
ReadLn;
End.

```

Функция NewSItem

Эта функция, реализованная в модуле *Dialogs* может быть использована для создания связанного списка строк. Пример построения связанного списка с помощью функции *NewSItem* показан ниже.

```

{////////////////////
NEWSITEM.PAS: Пример использования функции NewSItem
////////////////////}
Uses Dialogs;
Var
AList : PSItem;
TempPtr : PSItem;

Begin
AList := NewSItem('Item 1',
NewSItem('Item 2',

```

```

        NewSItem('Item 3',
        NewSItem('Item 4',
        nil))));

TempPtr := AList;
While TempPtr <> Nil Do
begin
    Writeln( TempPtr.Value );
    TempPtr := TempPtr.Next;
end;
Readln;
End.

```

Заключение

Правильное понимание основных принципов, заложенных в Turbo Vision, является залогом успешного использования этой библиотеки. Если по какой-либо причине вам непонятны действия, выполняемые тем или иным объектом - лучший источник информации это исходные тексты самой библиотеки. Экспериментируйте, пытайтесь локализовать проблему, изменить свойства того или иного объекта, одним словом, только практика поможет вам в успешном освоении Turbo Vision.

ГЛАВА 2. Базовые объекты: TApplication и TProgram

В этом разделе мы рассмотрим ряд основных объектов, которые обязательно присутствуют в любом Turbo Vision - приложении. Также рассматриваются объекты *TDesktop* и *TBackground*. Показывается, какие методы каждого из объектов необходимо переопределить для изменения внешнего вида/функциональности приложения, и приводятся примеры переопределения ряда методов рассматриваемых объектов.

TApplication: объект-приложение

В основе любого приложения, создаваемого с использованием Turbo Vision, лежат два объекта: объект *TApplication*, представляющий собой объект-приложение, и объект *TProgram*, представляющий собой объект-программу. Объект-приложение - это отображаемый объект, который, с одной стороны, управляет экраном, а с другой - предоставляет программе ядро управления событиями. Как отображаемый объект, объект *TApplication* является владельцем трех объектов: *TMenuBar*, *TStatusLine* и *TDesktop*. Каждый из этих объектов отвечает за свой участок экрана. Таким образом, с помощью этих объектов *TApplication* управляет всем экраном. Управление событиями происходит в методе *Run*, который является циклом обработки сообщений, аналогичным циклу, присутствующему в любом приложении для среды Microsoft Windows.

Каждое Turbo Vision-приложение строится по схеме:

Инициализация-Выполнение-Завершение

Каждой фазе соответствует определенный метод объекта *TApplication*. Фазе инициализации соответствует конструктор *Init*, фазе выполнения - метод *Run*, а фазе завершения - деструктор *Done*:

Init-Run-Done

Интересно отметить, что выполнение даже такой минимальной программы:

```

Program TV_DEMO;
uses App;
Var
  MyApp : TApplication;

Begin
  MyApp.Init;
  MyApp.Run;
  MyApp.Done;
End.

```

дает нам среду, "знающую", как реагировать на события и управлять отображаемыми объектами.

Инициализация

Обычно инициализация среды Turbo Vision происходит при вызове метода *Init* объекта, реализующего приложение. В приведенном выше примере это вызов *MyApp.Init*. Этот метод выполняет следующие действия (мы рассмотрим их подробно, так как это необходимо для понимания основных принципов работы Turbo Vision) :

- Инициализация системы управления памятью - *InitMemory* (модуль *Memory*).
Устанавливается обработчик ошибок по переполнению кучи, устанавливается размер резервной области (*safety pool*), равный 4092 байт, и резервная область располагается в памяти.
- Инициализация видеосистемы - *InitVideo* (модуль *Drivers*).
Проверяется текущий режим экрана, он сохраняется в переменной *StartupMode*, затем инициализируются переменные *ScreenMode*, *ScreenWidth* и *ScreenHeight*.
- Инициализация механизма обработки событий - *InitEvents* (модуль *Drivers*).
Создается очередь событий размером в 16 событий и инициализируется (устанавливается собственный) обработчик событий от "мыши".
- Инициализация обработчика системных ошибок - *InitSysError* (модуль *Drivers*).
Устанавливаются обработчики прерываний 09, 1B, 21, 23 и 24. Сохраняется состояние *CtrlBreak*.
- Инициализация системы поддержки протокола (модуль *HistList*).
Создается и инициализируется протокол размером в 1024 байта.

Инициализация объекта *TProgram*.

Переменной *Application* присваивается значение *@Self*. Вызывается метод *TProgram.InitScreen*: устанавливаются значения переменных *ShadowSize* и *ShowMarkers* в зависимости от значения переменной *ScreenMode*. Инициализируется объект *TGroup*. Создается группа размером 0,0, *ScreenWidth*, *ScreenHeight*.

Затем происходит инициализация объектов *TDeskTop*, *TStatusLine* и *TMenuBar*. По умолчанию создается рабочая область размером во весь экран минус строка для меню и строка состояния, пустая полоса меню и строка состояния, обрабатывающая команды Alt-X, F10, Alt-F3, F5, Ctrl-F5 и F6. Созданные объекты помещаются в группу.

Система готова к работе. Отметим, что объекты *TDeskTop*, *TStatusLine* и *TMenuBar* создаются на стадии инициализации объекта-приложения и при необходимости их изменения переопределяются соответствующие методы:

Метод	Переменная	Элемент
InitDeskTop	DeskTop	Рабочая область
InitStatusLine	StatusLine	Строка состояния
InitMenuBar	MenuBar	Полоса меню

Все три метода наследуются объектом *TApplication* (на основе которого создается наше приложение) и являются виртуальными. Таким образом, при их переопределении вызываются реальные методы, а не унаследованные.

Выполнение

Следующая фаза в работе нашей программы - фаза выполнения. Мы используем унаследованный от объекта *TApplication* метод *Run*. В свою очередь, объект *TApplication* наследует этот метод от объекта *TProgram* и не переопределяет его. Реализация этого метода (в объекте *TGroup*) представляет собой цикл обработки сообщений, состоящий из вызовов двух методов: *GetEvent* - получить событие и *HandleEvent* - обработать событие. Как мы увидим ниже, метод *HandleEvent* объекта *TGroup* вызывает соответствующие методы всех объектов, включенных в группу. Таким образом, чтобы изменить реакцию объекта на

события, необходимо переопределить его метод *HandleEvent*.

Обычно метод *Run* не переопределяется.

Завершение

На этой фазе происходит деинициализация всех объектов (вызываются деструкторы). Для этого используется метод *TApplication.Done*. В этом методе происходит вызов деструктора *TProgram.Done*, а также функций завершения работы подсистем:

- поддержки протокола - *DoneHistory*;
- обработки системных ошибок - *DoneSysError*;
- обработки событий - *DoneEvents*;
- поддержки отображения - *DoneVideo*;
- управления памятью - *DoneMemory*.

Цветовая палитра приложения

Приложение может использовать три типа палитры: цветную (*ApColor*), черно-белую (*ApBlackWhite*) или монохромную (*ApMonochrome*), которые устанавливаются при инициализации системы (метод *TProgram.InitScreen*). При необходимости возможно изменение текущей палитры приложения. Для это необходимо изменить значение переменной *AppPalette*. В приведенном примере показано, как изменить значение этого поля и выполнить перерисовку элементов приложения.

```
//////////////////////////////////////////////////////////////////
APP_PAL.PAS: Пример переключения палитры приложения
//////////////////////////////////////////////////////////////////
uses Dos, Objects, App, Dialogs, Views, Menus, Drivers, MsgBox, Memory;
```

```
Const
  cmPalette = 300;
Var
  R      : TRect;
Type
  TDAp = Object(TApplication)
    Dialog : PDialog;
    procedure InitStatusLine; virtual;
    procedure HandleEvent(var Event : TEvent); virtual;
  End;

Procedure TDAp.InitStatusLine;
Begin
  GetExtent(R);
```

```

R.A.Y := R.B.Y - 1;
StatusLine := New(PStatusLine, Init(R,
NewStatusDef(0, $FFFF,
NewStatusKey('~Alt-X~ Exit', kbAltX, cmQuit,
NewStatusKey('~Alt-P~ Palette', kbAltP, cmPalette,
Nil)),
));
End;

Procedure TDApp.HandleEvent;
Begin
inherited HandleEvent(Event);
if Event.What = evCommand Then
Begin
Case Event.Command of
cmPalette : Begin
Randomize;
AppPalette := Random(3);
DoneMemory;
Application.Redraw;
End;
else Exit;
End;
End;
ClearEvent(Event);
End;

Var
DApp : TDApp;

Begin
DApp.Init;
DApp.Run;
DApp.Done;
End.

```

Также можно реализовать специальную процедуру, выполняющую изменение палитры приложения:

```

////////////////////////////////////////////////////////////////
Переключение палитры приложения
////////////////////////////////////////////////////////////////
Procedure SetApPalette(NewPalette : Integer);
Begin
If NewPalette in [ApColor..ApMonochrome] then
Begin
AppPalette := NewPalette;
DoneMemory;
Application.Redraw;
End
End;

```

Параметрами этой функции могут быть предопределенные значения палитр: *ApColor*, *ApBlackWhite* или *ApMonochrome*.

В новой версии Turbo Vision несколько расширены свойства объекта *TApplication*. Ниже приводится описание новых методов данного объекта и примеры их использования.

Метод Cascade

Этот метод вызывает метод *Desktop.Cascade*, предварительно определяя область расположения окон (через вызов метода *GetTileRect*).

Метод DosShell

Этот метод позволяет вызывать копию *COMMAND.COM* (определяемую через переменную *COMSPEC*), выполняя все необходимые действия по завершению работы системы обработки ошибок, системы управления событиями, видео-системы и системы управления памятью. Вызывается процедура *WriteShellMsg*, позволяющая задать сообщение типа

Type EXIT to return...

По завершении работы с копией *COMMAND.COM* система возвращается в нормальное состояние. Происходит перерисовка экрана.

В приведенной ниже программе показано использование процедур *DosShell* и *WriteShellMsg*.

```
{-----  
Пример вызова копии COMMAND.COM в Turbo Vision 2.0  
-----}
```

```
uses App, Objects, Drivers, Menus, Views;
```

```
Type
```

```
TShellApp = Object(TApplication)
```

```
  procedure InitStatusLine; virtual;
```

```
  procedure WriteShellMsg; virtual;
```

```
End;
```

```
Procedure TShellApp.InitStatusLine;
```

```
{ Добавить команду вызова COMMAND.COM }
```

```
Var R : TRect;
```

```
Begin
```

```
  GetExtent(R);
```

```
  R.A.Y := R.B.Y - 1;
```

```

New(StatusLine, Init(R,
  NewStatusDef(0, $EFFF,
    NewStatusKey('~Alt-X~ Exit ', kbAltX, cmQuit,
      NewStatusKey('~Alt-S~ Shell', kbAltS, cmDosShell,
        Nil)),
    Nil)));
End;

Procedure TShellApp.WriteShellMsg;
{ Переопределить сообщение при вызове COMMAND.COM }
Begin
  PrintStr('Для возврата в программу, введите команду EXIT...');
End;

Var ShellApp: TShellApp;
Begin
  ShellApp.Init;
  ShellApp.Run;
  ShellApp.Done;
End.

```

Метод GetTileRect

По умолчанию этот метод вызывает метод *Desktop.GetExtent*, но может быть переопределен для указания новой области, в которой будут располагаться окна при выполнении методов *Tile* и *Cascade*. См. пример в разделе, посвященном объекту *TDesktop*.

Метод HandleEvent

Этот метод выполняет обработку трех команд:

Команда	Метод
cmCascade	Cascade
cmDosShell	DosShell
cmTile	Tile

Остальные команды обрабатываются методом *TProgram.HandleEvent*. В версии 1.0 метод *TProgram.HandleEvent* не переопределялся.

Метод Tile

Этот метод вызывает метод *Desktop.Tile*, предварительно определяя область расположения окон (через вызов метода *GetTileRect*).

Метод WriteShellMsg

По умолчанию этот метод выводит сообщение

Type EXIT to return...

Для изменения выводимого сообщения необходимо переопределить этот метод. Пример переопределения этого метода приведен ниже:

Procedure TShellApp.WriteShellMsg;

```
{ Переопределить сообщение при вызове COMMAND.COM }
Begin
  PrintStr('Для возврата в программу, введите команду EXIT...');
End;
```

Часы в приложении

Часто в комплексных приложениях могут потребоваться часы, которые будут отображать текущее время. Реализация часов - это хороший пример использования метода *TApplication.Idle*. Ниже приведен пример реализации объекта *TTVClock*, построенного на основе объекта *TView* и его использование в приложении.

```
{//////////////////////////////////////////}
TVCLOCK: Часы для Turbo Vision-приложений
{//////////////////////////////////////////}
```

Uses

Objects, App, Views, Drivers, DOS;

Type

```
PTVClock = ^TTVClock;
TTVClock = Object(TView)
  Hour,Min : Word;
  constructor Init(var Bounds : TRect);
  procedure Draw; Virtual;
  procedure Update; Virtual;
End;
TClockApp = Object(TApplication)
  Clock : PTVClock;
  constructor Init;
  procedure Idle; Virtual;
End;
```

Constructor TClockApp.Init;

Var

R : TRect;

Begin

Inherited Init;

```

GetExtent (R);
Dec (R.B.X);
R.A.X := R.B.X - 8;
R.B.Y := R.A.Y + 1;
Clock := New (PTVClock, Init (R));
Insert(Clock);
End;
Procedure TClockApp.Idle;
Begin
  If (Clock <> nil) then Clock^.Update;
  Inherited Idle;
End;

Constructor TTVClock.Init;
Begin
  Inherited Init(Bounds);
  Min := 99;
  Update;
End;
Procedure TTVClock.Draw;
Var
  B : TDrawBuffer;
  C : Word;
  H : Word;
  A, Suffix : String;
Begin
  Suffix := ' pm';
  H := Hour MOD 12;
  If (Hour < 12) then Suffix [2] := 'a';
  If (H = 0) then H := 12;
  Str ((H * 1000) + Min:5, A);
  A [3] := ':';
  A := A + Suffix;
  C := GetColor (1);
  MoveChar (B, ' ', C, Size.X);
  MoveStr (B, A, C);
  WriteLine (0, 0, Size.X, 1, B);
End;
Procedure TTVClock.Update;
Var
  H, M, S, T : Word;
Begin
  GetTime (H, M, S, T);
  If (Hour <> H) OR (Min <> M) Then
    Begin
      Hour := H; Min := M;
      DrawView;
    End;
End;

Var
  ClockApp : TClockApp;
Begin
  ClockApp.Init;
  ClockApp.Run;
  ClockApp.Done;
End.

```

Объект *TTVClock* имеет три метода: конструктор *Init* и методы *Draw* и *Update*. Вызов конструктора *Init* приводит к перерисовке (начальному отображению) часов. Метод *Draw*, который обязательно должен присутствовать у всех отображаемых объектов, предназначен для отрисовки часов на экране. В данной реализации используется англоязычный суффикс *AM/PM*, который может быть заменен, при необходимости. Отображение секунд не производится, так как для перерисовки содержимого часов используется метод *Idle* объекта *TApplication* и предполагается, что пользователь работает с приложением. Часы помещаются в полосу меню, но могут быть размещены и в строке состояния или в каком-либо другом объекте. Напомним, что метод *Idle* объекта *TApplication* вызывается в том случае, если в системе не происходит никаких действий.

TProgram: объект-программа

Объект *TProgram* - это шаблон для любого Turbo Vision-приложения. Он задает основные свойства своему наследнику, объекту *TApplication*. В большинстве случаев именно объект *TApplication* используется в качестве основы для создаваемого приложения. Объект *TProgram* - это группа, в которой находятся рабочая область, меню и строка состояния.

Рассмотрим ряд методов объекта *TProgram*, которые необходимы для понимания работы Turbo Vision.

Метод *GetEvent*

По умолчанию метод *TView.GetEvent* вызывает методы *GetEvent* своих владельцев. Так как объект *TProgram* (или *TApplication*) является владельцем всех отображаемых объектов, то каждый вызов метода *GetEvent* завершается вызовом метода *TProgram.GetEvent*. Этот метод проверяет, не существует ли события *evPending*, и, если такое событие существует, *GetEvent* возвращает его. Затем вызывается процедура *GetMouseEvent*, и, если нет событий от мыши, вызывается процедура *GetKeyEvent*. Если нет событий от клавиатуры, вызывается метод *Idle*. Метод *GetEvent* также перенаправляет события *evKeyDown* и *evMouseDown* объекту *TStatusLine*.

Метод HandleEvent

Все события обрабатываются методом *TGroup.HandleEvent*. При нажатии клавиш Alt-1..Alt-9 создается событие *evBroadcast* со значением команды *cmSelectWindowNum* и значением поля *InfoPtr* от 1 до 9. Эти события обрабатываются методом *HandleEvent* объекта *TWindow*. Также, обрабатывается команда *cmQuit*, при которой вызывается метод *EndModal(cmQuit)* и выполнение приложения завершается.

Метод Idle

Этот метод вызывается каждый раз, когда очередь событий пуста. По умолчанию вызывается метод *StatusLine.Update*, что позволяет строке состояния отобразить текущий справочный контекст. Если набор команд изменялся после последнего вызова метода *Idle*, то всем отображаемым объектам направляется команда *cmCommandSetChanged*.

При переопределении метода *Idle* необходимо вызывать метод *Inherited Idle*. Также необходимо следить за тем, чтобы выполняемые в этом методе действия не задерживали работу самого приложения в целом.

OutOfMemory

Этот метод вызывается методом *ValidView*, когда последний обнаруживает, что функция *LowMemory* вернула значение *True*. В этом случае необходимо выполнить какие-либо действия, например, уведомить пользователя о том, что не достаточно памяти для выполнения операции. Ниже приведен один из способов переопределения метода *OutOfMemory*:

```
Procedure TDemoApp.OutOfMemory;  
Begin  
  MessageBox('Не достаточно памяти для выполнения операции.',  
    Nil, mfError + mfOKButton);  
End;
```

По умолчанию метод *TProgram.OutOfMemory* не выполняет никаких действий.

Функция *LowMemory* возвращает значение *True* в том случае, если вызов конструктора объекта приведет к

выделению памяти из резервной области, размер которой определен переменной *LowMemSize*.

Метод *PutEvent*

По умолчанию метод *TView.PutEvent* вызывает методы *PutEvent* своих владельцев. Так как объект *TProgram* (или *TApplication*) является владельцем всех отображаемых объектов, то каждый вызов метода *PutEvent* завершается вызовом метода *TProgram.PutEvent*. Этот метод сохраняет копию записи типа *TEvent* в буфере, а следующий вызов метода *GetEvent* возвращает сохраненное событие (*Pending := Event*).

Изменения в Turbo Vision 2.0(Б)

В TurboVision 2.0 объект *TProgram* содержит три дополнительных метода, описание которых приводится ниже.

Метод *CanMoveFocus*

Этот метод непосредственно связан с введением объектов проверки ввода и позволяет определить, можно ли переместить фокус с текущего окна на следующее. Это возможно только в том случае, если введены допустимые данные (метод *Valid* вернул значение *True*).

Метод *ExecuteDialog*

Этот метод используется для более гибкого управления панелями диалога, позволяя автоматически устанавливать начальные данные и сохранять введенные данные по закрытии панели диалога. Необходимо использовать этот метод вместо вызова *Desktop^.ExecView*.

Метод *InsertWindow*

С помощью этого метода выполняются следующие действия: проверяется окно на допустимость (*ValidView*), вызывается метод *CanMoveFocus*, и, если этот метод возвращает значение *True*, окно отображается. Если по каким-либо причинам метод *CanMoveFocus* возвращает значение *False*, окно удаляется из памяти и не

отображается. Необходимо использовать этот метод вместо вызова *Desktop^.Insert*.

Итак, мы рассмотрели объекты *TApplication* и *TProgram*. Интересно отметить, что этот объект является отображаемым: *TApplication* - это группа, в которую включены такие интерфейсные объекты, как рабочая область, меню и строка состояния.

TDesktop: Рабочая область

Объект *TDesktop* выполняет очень важную роль в жизни Turbo Vision-приложения: он служит фоном для расположения остальных отображаемых интерфейсных элементов. Обычно, этот объект располагается на экране между строкой состояния и меню. Объект *TDesktop* является владельцем объекта *TBackground*, который задает цвет и шаблон заполнения фона рабочей области. Объект *TDesktop* умеет располагать окна внутри себя либо заполнением (метод *Tile*), либо перекрытием (метод *Cascade*).

Для изменения шаблона заполнения фона необходимо переопределить метод *TApplication.InitDesktop* и в нем выполнить следующие действия:

```
Procedure TDemoApp.InitDesktop;  
Begin  
  Inherited InitDesktop;  
  Desktop^.Background^.Pattern := '_';  
End;
```

Можно добиться различных привлекательных эффектов, если создать объект-наследник *TBackground* и творчески подойти к его методу *Draw*. В этом случае также необходимо создать объект-наследник *TDesktop*, переопределить у него метод *InitBackground*, а в методе *TDemoApp.InitDesktop* выполнить следующие действия:

```
Procedure TDemoApp.InitDesktop;  
Var  
  R : TRect;  
Begin  
  GetExtent(R);  
  R.Grow(0,-1);  
  Desktop := New(PNewDesktop, Init(R));  
End;
```

где *PNewDesktop* - указатель на объект *TNewDesktop*, который должен содержать изменения, описанные выше.

Законченный пример изменения шаблона заполнения фона приводится ниже.

```
/////////////////////////////////////////////////////////////////  
Пример изменения шаблона заполнения фона  
/////////////////////////////////////////////////////////////////}
```

```
uses Objects, Drivers, Views, App;
```

```
Type
```

```
PNewBackground = ^TNewBackground;
```

```
TNewBackground = Object(TBackground)
```

```
Text : TTitleStr;
```

```
constructor Init(var Bounds : TRect);
```

```
procedure Draw; virtual;
```

```
End;
```

```
PNewDesktop = ^TNewDesktop;
```

```
TNewDesktop = Object(TDesktop)
```

```
procedure InitBackground; virtual;
```

```
End;
```

```
TTestApplication = Object(TApplication)
```

```
procedure InitDesktop; virtual;
```

```
End;
```

```
Constructor TNewBackground.Init;
```

```
Begin
```

```
Inherited Init(Bounds, ' ');
```

```
{...}
```

```
Text := ' ';
```

```
While Length(Text) < SizeOf(TTitleStr) - 1 do
```

```
Text := Text + Chr(Random(255));
```

```
{...}
```

```
End;
```

```
Procedure TNewBackground.Draw;
```

```
Var
```

```
DrawBuffer : TDrawBuffer;
```

```
Begin
```

```
Randomize;
```

```
MoveStr(DrawBuffer, Text, Random(15));
```

```
WriteLine(0,0,Size.X,Size.Y,DrawBuffer);
```

```
End;
```

```
Procedure TNewDesktop.InitBackground;
```

```
Var
```

```
R : TRect;
```

```
Begin
```

```
GetExtent(R);
```

```
Background := New(PNewBackground,Init(R));
```

```
End;
```

```
Procedure TTestApplication.InitDesktop;
```

```
Var
```

```
R : TRect;
```

```
Begin
```

```
GetExtent(R);
```

```
R.Grow(0,-1);
```

```
Desktop := New(PNewDesktop,Init(R));
```

```
End;
```

```

Var
  TestApp : TTestApplication;
Begin
  TestApp.Init;
  TestApp.Run;
  TestApp.Done;
End.

```

Отметим, что, хотя при запуске такой программы мы не получим эффектного экрана, приведенный текст может служить шаблоном для создания собственных шаблонов фона.

Хотя более подробно палитры, используемые в Turbo Vision рассматриваются в главе 4, покажем на небольшом фрагменте, как изменить цвет фона.

```

Function TDemoApp.GetPalette;
Const MyColor: TPalette = CAppColor;
Begin
  MyColor[1] := #$7F; {Background}
  GetPalette := @MyColor;
End;

```

Первый элемент палитры приложения (*CAppColor*) отвечает за цвет фона (объект *TBackground*). В данном примере переопределяется метод *TApplication.GetPalette* и устанавливается цвет фона - ярко-белый на сером.

Изменения в Turbo Vision 2.0

Объект TDesktop

В новой версии Turbo Vision для объекта *TDesktop* добавлены два метода: *Load* и *Store*, а также поле *TileColumnsFirst*. Ниже приводится описание изменений и примеры использования.

Поле TileColumnsFirst

Значение этого поля позволяет установить расположение окон по команде *cmTile*: горизонтально или вертикально.

В приведенном ниже примере показано, как использовать это поле, а также метод *TApplication.GetTileRect*.

----- **TILEDEMO.PAS** Пример изменения режима расположения окон -----

uses App, Objects, Drivers, Menus, Views;

Const

cmTileMode = 100; {Изменить режим расположения}

cmNewWindow = 101; {Добавить окно}

Type

TMyApp = Object(TApplication)

procedure HandleEvent(var Event : TEvent); virtual;

procedure InitStatusLine; virtual;

procedure GetTileRect(var R : TRect); virtual;

procedure NewWindow;

procedure SetTileMode;

End;

Procedure TMyApp.InitStatusLine;

Var R : TRect;

Begin

GetExtent(R);

R.A.Y := R.B.Y - 1;

New(StatusLine, Init(R,

NewStatusDef(0, \$EFFF,

NewStatusKey('~Alt-X~ Exit ', kbAltX, cmQuit,

NewStatusKey('~Alt-W~ Window', kbAltW, cmNewWindow,

NewStatusKey('~Alt-T~ Tile ', kbAltT, cmTileMode,

Nil))),

Nil)));

End;

Procedure TMyApp.GetTileRect;

{Изменить область расположения окон}

Begin

Desktop~.GetExtent(R);

R.Grow(-1,-1);

End;

Procedure TMyApp.HandleEvent;

Begin

Inherited HandleEvent(Event);

if Event.What = evCommand then

case Event.Command of

cmTileMode : SetTileMode;

cmNewWindow : NewWindow;

end;

End;

Procedure TMyApp.NewWindow;

Var

R : TRect;

TheWindow : PWindow;

Begin

R.Assign(10,5,38,15);

New(TheWindow, Init(R, 'Window', wnNoNumber));

TheWindow~.Options := TheWindow~.Options OR ofTileable;

InsertWindow(TheWindow);

End;

```

Procedure TMyApp.SetTileMode;
Var R : TRect;
Begin
  With Desktop^ do
    Begin
      {Изменить режим расположения окон}
      TileColumnsFirst := Not(TileColumnsFirst);
      GetExtent(R);
      End;
    {
      Используем метод объекта TApplication,
      чтобы вызывался метод GetTileRect
    }
    Tile;
  End;

Var MyApp: TMyApp;
Begin
  MyApp.Init;
  MyApp.Run;
  MyApp.Done;
End.

```

Методы Load и Store

Методы *Load* и *Store* используются для загрузки и сохранения объекта *TDesktop*.

Метод *Load* вызывает конструктор *Load* объекта *TGroup*, затем восстанавливает фон (поле *Background*) с помощью метода *GetSubViewPtr*, а затем считывает содержимое поля *TileColumnsFirst*.

Метод *Store* вызывает метод *Store* объекта *TGroup*, затем сохраняет поле *Background* с помощью метода *PutSubViewPtr*, а после этого записывает содержимое поля *TileColumnsFirst*.

TStatusLine: Строка состояния

Объект *TStatusLine* отвечает за отображение и функционирование строки состояния, которая обычно занимает самую нижнюю строку экрана. Строка состояния выполняет двойную функцию: во первых, она предназначена для отображения краткой информации, во вторых, с ее помощью определяются команды, которые используются в приложении. Для отображения краткой информации необходимо переопределить метод *Hint* - функция *TStatusLine.Hint* и в зависимости от текущего контекста присваивать этой функции необходимые значения. В Turbo Vision 2.0 введено понятие стандартных

строк состояния, а также строк состояния, определяемых пользователем.

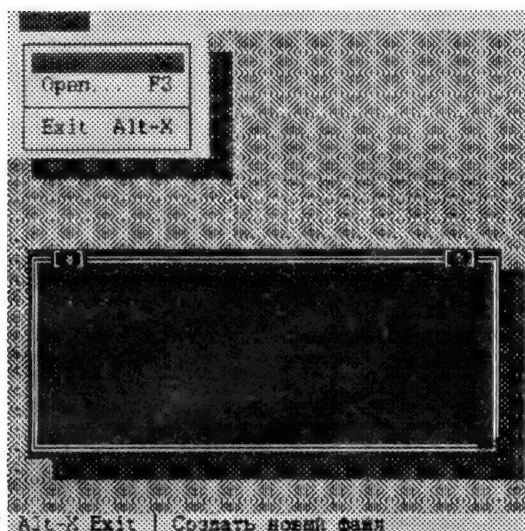


Рис.2.1.Подсказки в строке состояния.

```

/////////////////////////////////////////////////////////////////
HINTS.PAS: Пример переопределения метода Hint
/////////////////////////////////////////////////////////////////
uses Objects, Drivers, Menus, Views, App;
Const
  hcFile    = 1001;
  hcFileNew = 1002;
  hcFileOpen = 1003;
  hcFileExit = 1004;
  hcWindow  = 1005;
  cmFileNew = 100;
  cmFileOpen = 101;
Type
  PHintLine = ^THintLine;
  THintLine = Object(TStatusLine)
    Function Hint(AHelpCtx : Word) : String;   virtual;
  End;
  TDemoApp = Object(TApplication)
    constructor Init;
    procedure InitMenuBar;   virtual;
    procedure InitStatusLine; virtual;
  End;

Function THintLine.Hint;
Begin
  case AHelpCtx of
    hcFile    : Hint := 'Меню для работы с файлами';
  
```

```

hcFileNew : Hint := 'Создать новый файл';
hcFileOpen : Hint := 'Открыть существующий файл';
hcFileExit : Hint := 'Завершить выполнение приложения';
hcWindow : Hint := 'Это рабочее окно';
else
  Hint := '';
end;
End;

```

```

Constructor TDemoApp.Init;
Var
  R : TRect;
  Window : PWindow;
Begin
  Inherited Init;
  R.Assign(20,5,60,15);
  Window := New(PWindow, Init(R, 'wnNoNumber'));
  Window.HelpCtx := hcWindow;
  InsertWindow(Window);
End;

```

```

Procedure TDemoApp.InitMenuBar;
Var
  R : TRect;
Begin
  GetExtent(R);
  R.B.Y := R.A.Y + 1;
  MenuBar := New(PMenuBar, Init(R, NewMenu(
    NewSubMenu('F ile', hcFile, NewMenu(
      NewItem('N ew', 'F4', kbNoKey, cmFileNew, hcFileNew,
      NewItem('O pen...', 'F3', kbF3, cmFileOpen, hcFileOpen,
      NewLine(
        NewItem('E x it', 'Alt-X', kbAltX, cmQuit, hcFileExit,
        Nil)))))
    Nil))));
End;

```

```

Procedure TDemoApp.InitStatusLine;
Var
  R : TRect;
Begin
  GetExtent(R);
  R.A.Y := R.B.Y - 1;
  StatusLine := New(PHintLine, Init(R,
    NewStatusDef(0, $FFFF,
    NewStatusKey('Alt-X Exit', kbAltX, cmQuit,
    Nil),
    Nil)));
End;

```

```

Var
  DemoApp : TDemoApp;
Begin
  DemoApp.Init;
  DemoApp.Run;
  DemoApp.Done;
End.

```

Примечание: объект *TView*, являющийся предком всех отображаемых объектов, содержит поле контекста справочной системы - *HelpCtx*. Для задания этого контекста для любого отображаемого объекта необходимо просто присвоить значение этому полю:

```
Window := New(PWindow, Init(R, "wnNoNumber));
Window.HelpCtx := hcWindow;
InsertWindow(Window);
```

Отображение сообщений в строке состояния

Строку состояния можно использовать для временного отображения сообщений, которые должны привлечь внимание пользователя. Такой способ является альтернативой использованию функции *MessageBox*. В приведенном ниже примере показано, как реализовать отображение сообщений в строке состояния. Для этого я позаимствовал процедуру *SwapStatusLine* из модуля *Drivers*.

STAT_MSG.PAS: Отображение сообщений в строке состояния

```
uses Dos, Objects, App, Dialogs, Views, Menus, Drivers, MsgBox;
```

```
Const
  cmMessage = 300;
Var
  R : TRect;
Type
  TDAppl = Object(TApplication)
  Dialog : PDialog;
  procedure InitStatusLine; virtual;
  procedure HandleEvent(var Event : TEvent); virtual;
  procedure Message;
  procedure SwapStatusLine(var B);
End;
```

```
Procedure TDAppl.InitStatusLine;
Begin
  GetExtent(R);
  R.A.Y := R.B.Y - 1;
  StatusLine := New(PStatusLine, Init(R,
    NewStatusDef(0, $FFFF,
    NewStatusKey('Alt-X Exit', kbAltX, cmQuit,
    NewStatusKey('Alt-M Message', kbAltM,
    cmMessage,
    Nil)),
    Nil)
  ));
End;
```

```
{
  Копия процедуры DRIVERS.SwapStatusLine
```

```

}
Procedure TDAApp.SwapStatusLine(var B); Assembler;
asm
    MOV             CL,ScreenWidth
    XOR             CH,CH
    MOV             AL,ScreenHeight
    DEC             AL
    MUL             CL
    SHL             AX,1
    LES             DI,ScreenBuffer
    ADD             DI,AX
    PUSH            DS
    LDS             SI,B

@@@1:             MOV             AX,ES:[DI]
                 MOVSW
                 MOV             DS:[SI-2],AX
                 LOOP            @@@1
                 POP             DS

```

```

end;
Procedure TDAApp.Message;
Var
  C : Word;
  B : Array[0..79] of Word;
  S : String;
Begin
  C := $3E3F;
  S := '?! Ваше сообщение ?!';
  MoveChar(B, ' ', C, 80);
  MoveCStr(B[1], S, C);
  SwapStatusLine(B);
asm
  XOR AX,AX
  INT 16h
end;
SwapStatusLine(B);
End;

```

```

Procedure TDAApp.HandleEvent;
Begin
  inherited HandleEvent(Event);
  if Event.What = evCommand Then
    Begin
      Case Event.Command of
        cmMessage : Message;
      else Exit;
      End;
    End;
  ClearEvent(Event);
End;

```

```

Var
  DApp : TDAApp;

Begin
  DApp.Init;
  DApp.Run;
  DApp.Done;

```

End.

В методе *TDApp.Message* выполняется непосредственное отображение сообщения, которое создается с помощью процедуры *MoveCStr*. В качестве параметра *Attrs* можно задать необходимый цвет, которым будет выполняться отображение сообщения.

TMenuView: Полоса меню

Еще один объект - *TMenuView* - позволяет задать меню практически любого уровня вложенности. Меню определяется в методе *InitMenuBar* объекта-приложения. Создание меню достаточно подробно рассматривается в документации. Мы же рассмотрим ряд дополнительных возможностей.

Горизонтальное меню

Меню, не содержащее вложенных меню (разворачивающихся вниз), может быть создано, если вместо функции *NewSubMenu* использовать функцию *NewItem*. Создание такого меню показано в приведенном ниже примере.

```
Procedure TDemoApp.InitMenuBar;
Var
  R: TRect;
Begin
  GetExtent(R);
  R.B.Y := R.A.Y + 1;
  MenuBar := New(PMenuBar, Init(R, NewMenu(
    NewItem('F~ile', "", kbNoKey, cmCancel, hcNoContext,
    NewItem('E~dit', "", kbNoKey, cmCancel, hcNoContext,
    NewItem('S~earch', "", kbNoKey, cmCancel,
      hcNoContext,
    Nil))));
End;
```

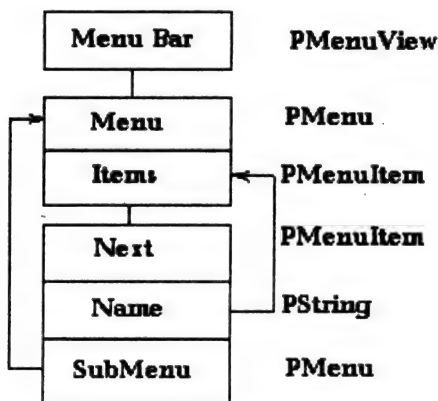
Созданное таким образом меню содержит только горизонтальные элементы, активация которых приводит к немедленному выполнению указанной команды.

Доступ к структуре меню

Как вы уже могли догадаться, меню хранятся в виде связанных списков. Таким образом, зная начальный

элемент, можно достигнуть любого элемента "дерева меню". Сначала рассмотрим структуры данных, реализующие иерархию меню.

Глобальная переменная *MenuBar* содержит указатель на основное меню приложения. Эта переменная имеет тип *PMenuView*. Объект *TMenuView* имеет два поля, которые могут быть интересны для решения нашей задачи: поле *ParentMenu* типа *PMenuView*, содержащее указатель на владельца данного меню, и поле *Menu* типа *PMenuView*, содержащее указатель на связанный список элементов меню. Структура *TMenu* содержит поле *Items* типа *PMenuItem*, представляющее собой указатель на один уровень дерева меню. В свою очередь, структура *TMenuItem* содержит полное описание элемента меню. Поясним сказанное диаграммой.



Как мы отметили выше, сначала необходимо найти вершину связанного списка. Согласно нашей диаграмме, вершиной дерева меню будет *MenuBar.Menu.Items*. Таким образом, чтобы обойти дерево по горизонтальным элементам, необходимо проверять поле *Next*. Для того, чтобы просмотреть вложенные меню, у каждого горизонтального элемента необходимо проверять поле *SubMenu*. В приведенном ниже примере показано, как обойти дерево меню, имеющее один уровень вложенности. Используя такую технику, можно реализовать обход дерева меню любого уровня вложенности.

```

Constructor TMenuTree.Init;
Var

```

```

R      : TRect;
MenuItem : PMenuItem;
SubMenu : PMenuItem;
I,J     : Integer;
Begin
R.Assign(0,0,50,20);
TWindow.Init(R,'Menu Tree',wnNoNumber);
Options := Options or ofCentered;
MenuItem := MenuBar^.Menu^.Items;
I := 1; J := 1;
While MenuItem <> Nil do
Begin
R.Assign(1,I,48,I+1);
Insert(New(PStaticText,Init(R,MenuItem^.Name)));
SubMenu := MenuItem^.SubMenu^.Items;
If SubMenu <> Nil Then
Begin
J := I+1;
While SubMenu <> Nil do
Begin
R.Assign(5,J,48,J+1);
If SubMenu^.Name <> Nil Then
Insert(New(PStaticText,Init(R,SubMenu^.Name)))
Else Insert(New(PStaticText,Init(R,'SEPARATOR')));
SubMenu := SubMenu^.Next;
J := J+1;
End;
End;
MenuItem := MenuItem^.Next;
I := J+1;
End;
End;

```

Примечание: в приведенном фрагменте программы создается окно, которое заполняется информацией о дереве меню. Например, может выводиться следующая информация:

```

F~ile
  N~ew
  O~pen...
  S~ave
  S~ave as...
  Save a~ll
  SEPARATOR
  C~hange dir...
  D~OS shell
  E~xit

E~dit
  U~ndo
  SEPARATOR
  Cu~t
  C~opy
  P~aste
  C~lear

```

Зная, как достичь любого элемента меню, можно реализовать функции работы с меню, аналогичные функциям Microsoft Windows API для добавления и удаления элементов меню, для динамического создания меню и подменю и т.п.

Интересной возможностью также является создание локальных меню. Такие меню создаются на основе объекта *TMenuPopUp*, реализованного в Turbo Vision версии 2.0 или объекта *TMenuBox*.

Объект TMenuPopUp

С помощью этого объекта реализуется "всплывающее" меню. Конструктор *Init* содержит параметры для указания размера и указателя на структуру типа *TMenu*. Использование этого объекта показано на приведенном ниже примере. По своей реализации объект *TMenuPopUp* является наследником объекта *TMenuBox*, а его конструктор схож с конструктором этого объекта, за исключением того, что параметр *AParentMenu* в данном случае равен *Nil*.

{-----
**Пример использования объекта TMenuPopUp
 в Turbo Vision 2.0**
 -----}

```
uses App, Objects, Drivers, Menus, Views;

Const
  cmMenu      = 100;          {Вызов меню}
  cmFileOpen  = 101;
  cmFileClose = 102;
Type
  TMenuApp = Object(TApplication)
    procedure HandleEvent(var Event : TEvent);   virtual;
    procedure InitStatusLine;                    virtual;
    procedure MenuPopUp;
  End;

Procedure TMenuApp.HandleEvent;
Begin
  Inherited HandleEvent(Event);
  If Event.What = evCommand Then
  Begin
    Case Event.Command of
      cmMenu : MenuPopUp;
    else
      Exit;
    End;
  ClearEvent(Event);
End;
```

End;

```
Procedure TMenuApp.InitStatusLine;  
{ Добавить команду вызова меню }  
Var R : TRect;
```

Begin

GetExtent(R);

R.A.Y := R.B.Y - 1;

New(StatusLine, Init(R,

NewStatusDef(0, \$EFFF,

NewStatusKey('~Alt-X~ Exit ', kbAltX, cmQuit,

NewStatusKey('~Alt-M~ Menu', kbAltM, cmMenu,

Nil)),

Nil)));

End;

```
Procedure TMenuApp.MenuPopUp;
```

Var

R : TRect;

MP : PMenuPopUp;

Begin

R.Assign(10, 5, 20, 15);

MP := New(PMenuPopUp, Init(R,

NewMenu(

NewItem('~O~pen', 'F3', kbF3, cmFileOpen, hcNoContext,

NewItem('~C~lose', 'Alt-F3', kbAltF3, cmFileClose,

hcNoContext,

Nil)))

));

Desktop~.Insert(MP);

End;

```
Var MenuApp : TMenuApp;
```

Begin

MenuApp.Init;

MenuApp.Run;

MenuApp.Done;

End.

Используя объект *TMenuBox*, метод *TMenuApp.MenuPopUp* может быть реализован следующим образом:

```
Procedure TMenuApp.MenuPopUp;
```

Var

R : TRect;

MP : PMenuBox;

Begin

R.Assign(10, 5, 20, 15);

MP := New(PMenuBox, Init(R,

NewMenu(

NewItem('~O~pen', 'F3', kbF3, cmFileOpen, hcNoContext,

NewItem('~C~lose', 'Alt-F3', kbAltF3, cmFileClose, hcNoContext,

Nil))),

Nil));

Desktop~.Insert(MP);

End;

Как поместить элемент меню в самую левую позицию

В приведенном ниже примере показано, как поместить вновь созданный элемент меню в самую левую позицию. Такой способ может быть полезен при создании временного меню, например, при отладке программ. Зная структуру меню, реализация такой задачи выполняется довольно просто. Сначала выполняется проверка на наличие меню:

```
if (MenuBar <> Nil) ...
```

и если меню уже существует, новый элемент помещается в него. В случае отсутствия полосы меню она сначала создается, а затем в нее помещается необходимый элемент.

{///
Поместить элемент меню AMenu в самую левую позицию
///}

```
Procedure Menu1St(AMenu : PMenuItem);
Begin
  if (MenuBar <> nil) then
    begin
      if (MenuBar^.Menu <> nil) then
        begin
          AMenu^.Next := MenuBar^.Menu^.Items;
          MenuBar^.Menu^.Items := AMenu;
        end
      else
        MenuBar^.Menu := NewMenu(AMenu);
      end
    else
      begin
        GetExtent(Rect);
        Rect.B.Y := Rect.A.Y + 1;
        MenuBar := New(PMenuBar,
          Init(Rect, NewMenu(AMenu)));
      end;
    End;
```

Как сделать меню с маркерами

В ряде приложений бывает удобно устанавливать специальные отметки для отдельных элементов меню. Такие отметки мы будем называть маркерами. В этом разделе рассматривается один из способов реализации меню подобного типа. Идеально было бы переопределить тип данных *TMenuItem* и добавить новое поле, в котором бы содержалось текущее значение маркера: отображается он

или нет. К сожалению, система меню реализована не в качестве объекта, а как связанный список: расширение функциональности системы меню потребовало бы изменения многих других фрагментов исходного кода. Вместо этого предлагается дополнительная функция *NewCheckItem*, которая автоматически оставляет место для маркера в каждом элементе меню. Если параметр *Checked* этой функции равен *True*, отображается маркер, для которого выбран символ '№' (код 251). Символ для маркера может быть изменен, если переопределить значение константы *CheckMark*. Также, реализованы четыре функции для управления состоянием элемента меню.

Функция	Назначение
CheckMenuItem	Помещает маркер в элемент меню
ClearMenuItem	Снимает отметку с элемента меню
MenuItemIsChecked	Возвращает состояние элемента меню
ToggleMenuItem	Изменяет состояние элемента меню

Каждая из этих функций использует два параметра:

Параметр	Описание
AMenu : PMenuItem	Обычно это поле Menu глобальной переменной MenuBar
Command : Word	Команда типа cmXXX для данного элемента меню

Ниже приводится исходный текст модуля *MenuMark.Pas*, в котором реализованы описанные выше функции.

```
{//////////////////////////////////////
MENUMARK: Модуль для реализации маркеров
//////////////////////////////////////}

Unit MenuMark;

Interface
uses Menus;

Function NewCheckItem(Name, Param      : TMenuStr;
                      KeyCode, Command : Word;
                      AHelpCtx         : Word;
                      Checked           : Boolean;
                      Next              : PMenuItem ) : PMenuItem;

Function CheckMenuItem(AMenu      : PMenu;
```

```

        var Command : Word) : Boolean;
Function ClearMenuItem(AMenu      : PMenu;
        var Command : Word) : Boolean;
Function MenuItemIsChecked(AMenu  : PMenu;
        var Command : Word) : Boolean;
Function ToggleMenuItem(AMenu     : PMenu;
        var Command: Word) : Boolean;

```

Const

```

    CheckMark : Char = 'N';
    ClearMark  : Char = ' ';

```

Implementation

```

Function NewCheckItem(Name, Param : TMenuStr; KeyCode, Command
:Word; AHelpCtx : Word; Checked : Boolean; Next : PMenuItem) : PMenuItem;
Begin
    if Name <> "" then
    begin
        Name := ' ' + Name;
        if Checked then
            Name := CheckMark + Name else
            Name := ClearMark + Name;
        end;
        NewCheckItem := NewItem(Name, Param, KeyCode, Command, AHelpCtx,
Next);
    End;

```

```

Function FindMenuItem(AMenu : PMenu; Command : Word) : PMenuItem;

```

```

var
    P, Q : PMenuItem;
begin
    P := AMenu^.Items;
    while P <> Nil do
    begin
        if (P^.Command = 0) and (P^.Name <> Nil) then
        begin
            Q := FindMenuItem(P^.SubMenu, Command);
            if Q <> Nil then
            begin
                FindMenuItem := Q;
                Exit;
            end;
        end else
        begin
            if (P^.Command = Command) and not P^.Disabled
            then
            begin
                FindMenuItem := P;
                Exit;
            end;
        end;
        P := P^.Next;
    end;
    FindMenuItem := Nil;
end;

```

```

Function CheckMenuItem(AMenu : PMenu; var Command : Word) : Boolean;
var
    MenuItem : PMenuItem;

```

```

begin
  CheckMenuItem := False;
  MenuItem := FindMenuItem(AMenu, Command);
  if MenuItem <> Nil then
    begin
      if MenuItem^.Name^[1] = ClearMark then
        begin
          MenuItem^.Name^[1] := CheckMark;
          CheckMenuItem := True;
        end;
      end else Command := 0;
    end;
end;

```

Function ClearMenuItem(AMenu : PMenu; var Command : Word) : Boolean;

```

var
  MenuItem : PMenuItem;
begin
  ClearMenuItem := False;
  MenuItem := FindMenuItem(AMenu, Command);
  if MenuItem <> Nil then
    begin
      if MenuItem^.Name^[1] = CheckMark then
        begin
          MenuItem^.Name^[1] := ClearMark;
          ClearMenuItem := True;
        end;
      end else Command := 0;
    end;
end;

```

Function MenuItemIsChecked(AMenu : PMenu; var Command : Word) : Boolean;

```

var
  MenuItem : PMenuItem;
begin
  MenuItemIsChecked := False;
  MenuItem := FindMenuItem(AMenu, Command);
  if MenuItem <> Nil then
    begin
      if MenuItem^.Name^[1] = CheckMark then
        MenuItemIsChecked := True;
      end else Command := 0;
    end;
end;

```

Function ToggleMenuItem(AMenu : PMenu; var Command : Word) : Boolean;

```

begin
  if MenuItemIsChecked(AMenu, Command) then
    ClearMenuItem(AMenu, Command) else
    if Command <> 0 then CheckMenuItem(AMenu,
Command);
end;
end.

```

Далее приведем пример использования функций, реализованных в модуле MenuMark.Pas.

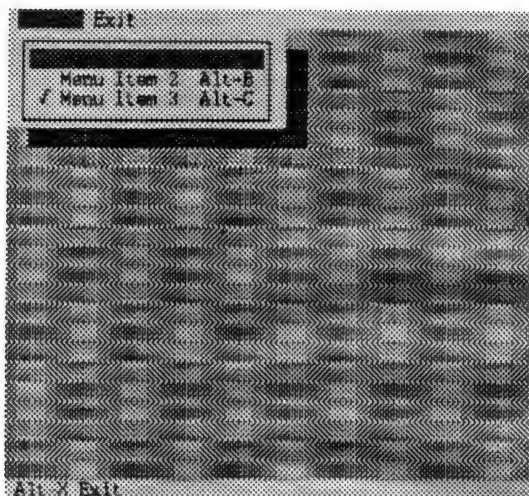


Рис. 2.2. Меню с пометками

```

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
MMTEST.PAS: Пример использования модуля MENUMARK
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

```

```

uses App, Drivers, Menus, MenuMark, Objects, Views;

```

```

const

```

```

    cmMenuItem1 = 100;
    cmMenuItem2 = 101;
    cmMenuItem3 = 102;

```

```

type

```

```

    PMyApp = ^TMyApp;
    TMyApp = object(TApplication)
        Procedure HandleEvent(var Event : TEvent); virtual;
        Procedure InitMenuBar; virtual;
    End;

```

```

Procedure TMyApp.HandleEvent(var Event : TEvent);

```

```

Begin

```

```

    if (Event.What = evCommand) then
    if (Event.Command in [cmMenuItem1..cmMenuItem3]) then
    begin

```

```

        if MenuItemIsChecked(MenuBar^.Menu, Event.Command)
        then

```

```

            ClearMenuItem(MenuBar^.Menu, Event.Command)

```

```

        else

```

```

            CheckMenuItem(MenuBar^.Menu, Event.Command);

```

```

    end;

```

```

    TApplication.HandleEvent(Event);

```

```

End;

```

```

Procedure TMyApp.InitMenuBar;

```

```

var
  R : TRect;
Begin
  GetExtent(R);
  R.B.Y := R.A.Y + 1;
  MenuBar := New(PMenuBar, Init(R, NewMenu(
    NewSubMenu('T~est', hcNoContext, NewMenu(
      NewCheckItem('Menu Item ~1~', 'Alt-A', kbAltA,
        cmMenuItem1, hcNoContext, False,
      NewCheckItem('Menu Item ~2~', 'Alt-B', kbAltB,
        cmMenuItem2, hcNoContext, False,
      NewCheckItem('Menu Item ~3~', 'Alt-C', kbAltC,
        cmMenuItem3, hcNoContext, True,
        nil))),
    NewItem('E x ~it', "", kbNoKey, cmQuit, hcNoContext, nil))));
End;

Var
  MyApp : TMyApp;

Begin
  MyApp.Init;
  MyApp.Run;
  MyApp.Done;
End.

```

Расширения

Одно из преимуществ применения технологии объектно-ориентированного программирования заключается в возможности расширения свойств объектов. В этом разделе мы рассмотрим ряд практических примеров расширения функциональности Turbo Vision.

Использование справочного контекста

Выше уже упоминалось использование справочного контекста совместно с методом *Hint*. Помимо этого, имеется возможность управления всей строкой состояния с помощью значений справочного контекста. Для этого необходимо выбрать диапазоны справочного контекста. Значения могут изменяться от 0 до \$FFFF. В качестве примера будем использовать три диапазона:

0..\$AFFF	"Стандартный диапазон"
\$B000..\$C000	Поддиапазон 1
\$C000..\$F000	Поддиапазон 2

Для диапазона \$B000..\$C000 создадим функцию *StatusSet1*, с помощью которой задаются команды, скажем, перехода к следующему и предыдущему полям (клавиши F6 и *Shift F6*). Инициализируем строку состояния нашего приложения (объект *TMyApp*). Теперь самое простое - изменить в необходимый момент значение контекста. Это изменение выполняется при отображении панели диалога. Обратите внимание, что при появлении панели диалога отображается набор функций для работы с этой панелью (обработчик и сами функции в примере не реализованы). После завершения работы с панелью (при ее закрытии) набор команд в строке состояния изменяется. Пример использования диапазонов справочной системы для изменения допустимых команд в строке состояния показан ниже.

```
{-----  
Пример использования диапазонов справочной  
системы для изменения строки состояния  
-----}
```

```
Uses Objects,App,Drivers,Menus,Views,Dialogs;
```

```
Const
```

```
  cmDlg      = 100;  
  cmNextFld  = 200;  
  cmPrevFld  = 201;
```

```
Type
```

```
  TMyApp = Object(TApplication)  
    procedure HandleEvent(var Event : TEvent); virtual;  
    procedure InitStatusLine; virtual;  
    procedure NewDialog;  
  End;
```

```
Function StatusSet1(Next : PStatusItem) : PStatusItem;
```

```
Begin
```

```
  StatusSet1 :=  
    NewStatusKey('~F6~ NextFld',kbF6,cmNextFld,  
    NewStatusKey('~ShiftF6~ PrevFld',  
      kbShiftF6,cmPrevFld,  
    Next));
```

```
End;
```

```
Procedure TMyApp.InitStatusLine;
```

```
Var
```

```
  R : TRect;
```

```
Begin
```

```
  GetExtent(R);  
  R.A.Y := R.B.Y - 1;  
  New(StatusLine,Init(R,  
    NewStatusDef(0,$AFFF, {Стандартный диапазон}  
    StdStatusKeys(  
    NewStatusKey('~Alt-D~ Dialog',kbAltD,cmDlg,Nil)).
```

```

NewStatusDef($B000,$C000,      {Поддиапазон 1}
  StatusSet1(Nil),
NewStatusDef($C000,$F000,      {Поддиапазон 2}
  StdStatusKeys(Nil),
  Nil)))));
End;

Procedure TMyApp.HandleEvent;
Begin
  Inherited HandleEvent(Event);
  If Event.What = evCommand then
    Case Event.Command of
      cmDlg : NewDialog;
    End;
End;

Procedure TMyApp.NewDialog;
Var
  R : TRect;
  Dlg : PDialog;
Begin
  R.Assign(0,0,40,12);
  New(Dlg,Init(R,'Demo Dialog'));
  Dlg.Options := Dlg.Options OR ofCentered;
  Dlg.HelpCtx := $B000;
  Insert(Dlg);
End;

Var
  MyApp : TMyApp;

Begin
  MyApp.Init;
  MyApp.Run;
  MyApp.Done;
End.

```

Отметим, что приведенный выше пример показывает, что, управляя контекстом справочной системы, можно задавать уникальные клавиши для определенных объектов. Все, что необходимо сделать для переключения контекста, - это присвоить ему новое значение:

```
Object.HelpCtx := Значение;
```

перед тем как объект будет активизирован (в случае панели диалога - *Insert(Dialog)*).

О сохранении экрана

Каждый, кто пользовался интегрированной средой разработчика, наверняка, и не раз, нажимал клавишу AltF5

для просмотра результатов работы программы. Почему бы не реализовать эту возможность в своем приложении? Давайте посмотрим, как это можно сделать.

Создадим объект *TVideoObj*, который будет обладать следующими свойствами: уметь определять адрес видеобуфера, сохранять и восстанавливать содержимое видеобуфера,

```
TVideoObj = Object(TObject)
Function VideoMemory : LongInt;
Procedure SaveScreen;
Procedure RestoreScreen;
End;
```

Затем заведем глобальную переменную *Screen*:

```
Var Screen : Pointer;
```

Теперь мы готовы к реализации методов объекта *TVideoObj*. Метод *VideoMemory* возвращает адрес видеобуфера в зависимости от текущего режима адаптера. Отметим, что использование предопределенных переменных *SegXXX* возможно только в версии компилятора 7.0, это позволяет программе работать в защищенном режиме.

```
Function TVideoObj.VideoMemory : LongInt;
Begin
  {$IFDEF VER70}
    VideoMemory := SegB800;
    If Mem[Seg0040:0049] = 7 Then VideoMemory := SegB000;
  {$ENDIF}
  {$IFDEF VER60}
    VideoMemory := $B800;
    If Mem[$0040:$0049] = 7 Then VideoMemory := $B000;
  {$ENDIF}
End;
```

Методы для сохранения/восстановления содержимого видеобуфера могут быть реализованы следующим образом (отметим, что реализацию для режима 43/50 строк мы оставляем читателям в качестве упражнения):

```
Procedure TVideoObj.SaveScreen;
Begin
  GetMem(Screen, 4000);
  Move(Mem[VideoMemory : 0], Screen, 4000);
End;
```

```
Procedure TVideoObj.RestoreScreen;
Begin
  If Screen = Nil Then Exit;
```

```

Move(Screen^,Mem[VideoMemory : 0],4000);
End;

```

§ 2

Сохранение экрана выполняется перед инициализацией системы в конструкторе объекта *TMyApp*, который является нашим "приложением". Еще одна тонкость заключается в том, что перед вызовом метода *Redraw* мы вызываем процедуру *DoneMemory* для очистки всех буферов в памяти. Пример реализации функции просмотра пользовательского экрана с вызовом по клавише Alt-F5 показан ниже.

Пример реализации функции просмотра пользовательского экрана. Вызов - клавиша AltF5

```

{-----}
{$X+}
Uses
  Dos,Crt,App,Views,Objects,Drivers,Memory,Menus;

Const
  cmUserScreen = 1000;
Var
  Screen : Pointer;

Type

  TVideoObj = Object(TObject)
    Function VideoMemory : LongInt;
    Procedure SaveScreen;
    Procedure RestoreScreen;
  End;

  TMyApp = Object(TApplication)
    V : TVideoObj;
    Constructor Init;
    procedure InitStatusLine; virtual;
    procedure HandleEvent(Var Event : TEvent); virtual;
    procedure UserScreen;
  End;

Function TVideoObj.VideoMemory : LongInt;
Begin
  {-----}
  Определить адрес видеобuffers
  {-----}
  VideoMemory := SegB800;
  If Mem[Seg0040:$0049] = 7 Then VideoMemory := SegB000
End;

Procedure TVideoObj.SaveScreen;
Begin
  {-----}
  Сохранить содержимое видеобuffers.
  Вариант для режима 80x25
  {-----}

```

```

GetMem(Screen,4000);
Move(Mem[VideoMemory : 0],Screen^,4000);
End;

Procedure TVideoObj.RestoreScreen;
Begin
{-----}
Восстановить содержимое видеобuffers.
Вариант для режима 80x25
{-----}
If Screen = Nil Then Exit;
Move(Screen^,Mem[VideoMemory : 0],4000);
End;

Constructor TMyApp.Init;
Begin
Screen := Nil;
V.SaveScreen; {Сохранить экран перед инициализацией}
Inherited Init; {Инициализация}
End;

Procedure TMyApp.InitStatusLine;
Var
R : TRect;
Begin
GetExtent(R);
R.A.Y := R.B.Y - 1;
StatusLine := New(PStatusLine,Init(R,
NewStatusDef(0,$FFFF,
NewStatusKey('~Alt-X~ Exit', kbAltX,cmQuit,
NewStatusKey('~Alt-F5~ User
Screen^,kbAltF5,cmUserScreen,
Nil)),
Nil)
));
End;

Procedure TMyApp.HandleEvent;
Begin
Inherited HandleEvent(Event);
If Event.What = evCommand then
Begin
Case Event.Command of
cmUserScreen : UserScreen; {Показать экран
пользователя}

Else
Exit;
End;
ClearEvent(Event);
End;
End;

Procedure TMyApp.UserScreen;
Begin
HideMouse; {Курсор "мыши" не отображается}
V.RestoreScreen; {Восстановить экран}
ReadKey; {Ждать нажатия любой клавиши}
DoneMemory; {Очистить buffers}

```

Redraw;	{Перерисовать все}
ShowMouse;	{Отобразить курсор "мышь"}
End;	
Var	
MyApp : TMyApp;	
Begin	
MyApp.Init;	
MyApp.Run;	
MyApp.Done;	
End.	

Программы без меню

В ряде приложений может возникнуть необходимость расширения размера рабочей области. В этом случае мы можем отказаться от полосы меню как источника команд. Изначально, если меню не инициализировано, т.е. у объекта "приложение" не переопределен метод *InitMenuBar*, мы получаем пустое меню, которое тем не менее отображается на экране. Происходит это из-за того, что метод *InitMenuBar* объекта *TProgram* создает "пустое" меню:

```
MenuBar := New(PMenuBar, Init(R, Nil));
```

Что нужно сделать, чтобы меню не отображалось? Метод *TProgram.Init* вызывает свой (или переопределенный) метод *InitMenuBar* и в случае, если переменная *MenuBar* не равна *Nil*, активизирует созданный объект:

```
Insert(MenuBar)
```

Таким образом, если в переопределенном методе *InitMenuBar* выполнить присвоение:

```
MenuBar := Nil
```

полоса меню отображаться не будет. Пример программы, не использующей меню, показан ниже.

```
{-----}
Пример программы, не использующей меню.
Полоса меню не отображается
{-----}
```

```
uses Objects, Drivers, Views, App;
Type
  TMyApp = Object(TApplication)
    Procedure InitDeskTop;           virtual;
    Procedure InitMenuBar;          virtual;
```

```

End;

Procedure TMyApp.InitDeskTop;
Var
  R : TRect;
Begin
  GetExtent(R);
  R.A.Y := 0;
  Desktop := New(PDesktop, Init(R));
End;

Procedure TMyApp.InitMenuBar;
Begin
  MenuBar := Nil;
End;

Var
  MyApp: TMyApp;

Begin
  MyApp.Init;
  MyApp.Run;
  MyApp.Done;
End.

```

Отметим, что в методе *InitDeskTop* нам необходимо увеличить размер рабочей области перед ее инициализацией.

Скрытая строка состояния

Далее мы можем выиграть еще одну строку за счет того, что после инициализации строка состояния не будет отображаться на экране. Отметим, что функциональность такой строки сохраняется полностью - она реагирует на нажатия клавиш и посылает команды.

{-----
Пример программы с неотображаемой строкой состояния
-----}

```

uses Objects, Drivers, Views, App;
Type
  TMyApp = Object(TApplication)
    Procedure InitDeskTop;           virtual;
    Procedure InitStatusLine;       virtual;
  End;

Procedure TMyApp.InitDeskTop;
Var
  R : TRect;
Begin
  GetExtent(R);
  R.A.Y := 0;
  Desktop := New(PDesktop, Init(R));

```

```

End;

Procedure TMyApp.InitStatusLine;
Begin
  Inherited InitStatusLine;
  StatusLine^.SetState(sfVisible,False);
End;

Var
  MyApp : TMyApp;

Begin
  MyApp.Init;
  MyApp.Run;
  MyApp.Done;
End.

```

Все, что необходимо сделать в данном случае, - это переопределить метод *InitStatusLine* (*TMyApp.InitStatusLine*), создать строку состояния (в данном примере используется строка состояния создаваемая по умолчанию - *Inherited InitStatusLine*), а затем обнулить значение флага *sfVisible*:

```
StatusLine^.SetState(sfVisible,False)
```

Отметим, что, управляя флагом *sfVisible*, мы можем получить эффект переключаемой строки состояния.

Режим 132x25

Еще один способ размещения большого объема информации на экране может быть основан на том факте, что видеоадаптеры VGA могут работать в режимах высокого разрешения. В качестве примера такого подхода можно указать среду СУБД Paradox 4.0 фирмы Borland. Для реализации нам необходимо знать, каким способом тот или иной видеоадаптер переключается в режим высокого разрешения.

В приведенной ниже таблице показаны значения регистров при вызове прерывания Int 10, используемые для переключения видеоадаптера в режим высокого разрешения.

Видеоадаптер	Режим переключения
Ahead V5000	AL = 23h
ATI	AL = 23h
Chips & Technologies	AL = 60h
Cirrus Logic	AL = 15h

Everex	AX = 0070h/BL = 0Ah
Genoa	AL = 60h
OAK OTI-067	AL = 50h
Paradise	AL = 55h
Trident	AL = 53h
Tseng	AL = 23h
Vesa	AX = 4F02h/BX = 109h
Video7	AX = 6F05h/BL = 41h

Затем необходимо переопределить метод *InitScreen*. Этот метод вызывается при начальной инициализации, а также при любых изменениях, связанных с видеорежимом.

{-----
**Пример запуска TurboVision-программы
 в режиме высокого разрешения: 132x25**
 -----}

```
{X+}
Uses
  Objects, Drivers, Views, App, Dialogs;

Type
  TMyApp = Object(TApplication)
    Procedure InitScreen; virtual;
  End;

Procedure TMyApp.InitScreen;
Var
  R : TRect;
Begin
  Inherited InitScreen;
  {Установка режима 132x25 для видеоадаптера Video7}
  ASM
    MOV AX, $6F05
    MOV BL, $41
    INT 10H
    MOV ScreenMode, $41; {Видеорежим}
  End;
  ScreenHeight := 25;      {Высота экрана}
  ScreenWidth  := 132;     {Ширина экрана}
End;

Var
  MyApp: TMyApp;

Begin
  MyApp.Init;
  MyApp.Run;
  MyApp.Done;
End.
```

Применение описанных выше изменений не ограничивается режимом 132x25. Наиболее правильным в этом случае является создание специального файла,

доступного программе, в котором были бы описаны возможные нестандартные режимы для некоторых наиболее распространенных видеокарт. Содержимое такого файла могло бы отображаться в специальном окне (вызываемом по команде *Options*), где содержались бы возможные режимы работы для данного типа видеокарты.

Динамически изменяемые меню и строки состояния

гот
же

Используя ту же технику, что и при замене рабочей области, мы можем создавать динамически изменяемые меню. Это может быть удобно при создании программ, ориентированных на многоязычные пользовательские интерфейсы (русский/английский и т.п.), а также для реализации "многоуровневых" программ. В последнем случае может существовать меню для начинающих пользователей, пользователей среднего уровня и для опытных пользователей. Такой уровеньный подход может быть также использован при создании демонстрационных версий программ. В этом случае используется меню, не включающее ряд опций.

Как известно, меню создается с помощью метода *InitMenuBar* объекта -наследника объекта *TApplication*. Создание меню показано в приведенном ниже фрагменте:

```
procedure TMyApp.InitMenuBar;
Var
  R : TRect;
Begin
  .....

  MenuBar := New(PMenuBar, Init(R, NewMenu(
    NewSubMenu(
      NewItem
    .....
  End;
```

Как видно из этого фрагмента, в методе *InitMenuBar* создается новый объект типа *TMenuBar* (при вызове функции *New* вызывается его конструктор *Init*), экземпляр которого называется *MenuBar*. Как известно, все, что создается с помощью вызова конструктора, может быть удалено с помощью деструктора. Таким образом, мы можем вызвать деструктор *Done* и тем самым завершить существование экземпляра этого объекта. Последующий вызов конструктора позволит нам создать новый объект

этого типа, но с измененными свойствами. Поясним сказанное на примере.

Динамически изменяемое меню

```
{-----  
DYNAMENU.PAS. Динамически изменяемое меню  
-----}
```

Uses

Objects,	{используем TRect}	
Menus,	{используем группу объектов поддержки меню}	
Drivers,	{используем обработчик событий}	}
Views,	{используем справочную систему}	
App;	{используем объект TApplication}	}

Const

cmSwitchMenu = 100; {Команда переключения меню}

Type

TMyApp = Object(TApplication)

cMenu : Boolean; {тип меню}

constructor Init;

procedure InitRMenu; {русскоязычное меню}

procedure InitEMenu; {англоязычное меню}

procedure InitMenuBar; virtual;

procedure SwitchMenu; {метод переключения меню}

procedure HandleEvent(var Event : TEvent); virtual;

End;

Constructor TMyApp.Init;

Begin

TApplication.Init;

cMenu := False;

End;

Procedure TMyApp.InitEMenu;

{Англоязычное меню}

Var

R : TRect;

Begin

GetExtent(R);

R.B.Y := R.A.Y + 1;

MenuBar := New(PMenuBar, Init(R, NewMenu(
NewItem("M"enu", kbNoKey, cmSwitchMenu, hcNoContext,
nil))));

End;

Procedure TMyApp.InitRMenu;

{Русскоязычное меню}

Var

R : TRect;

Begin

GetExtent(R);

```

R.B.Y := R.A.Y + 1;
MenuBar := New(PMenuBar, Init(R, NewMenu(
 NewItem("Меню", kbNoKey, cmSwitchMenu, hcNoContext,
  nil))));
End;

```

```

Procedure TMyApp.InitMenuBar;
Begin
  InitEMenu;
End;

```

```

Procedure TMyApp.SwitchMenu;
{Метод-переключатель меню}
Begin
  {Удалить текущее меню}
  Delete(MenuBar);
  Dispose(MenuBar, Done);

```

```

If cMenu then InitEMenu else InitRMenu;

```

```

cMenu := Not(cMenu); {Переключить тип меню}

```

```

{Отобразить новое меню}
Insert(MenuBar);
End;

```

```

Procedure TMyApp.HandleEvent;
{Обработчик событий}
Begin
  TApplication.HandleEvent(Event);
  if Event.What = evCommand then
  Begin
    case Event.Command of
      cmSwitchMenu : SwitchMenu;
    end;
  End;
  ClearEvent(Event);
End;

```

```

Var
  MyApp : TMyApp;

```

```

Begin

```

```

  MyApp.Init;
  MyApp.Run;
  MyApp.Done;

```

```

End.

```

Приведенный пример содержит минимальный код, необходимый для пояснения используемой техники. Сначала мы создаем два метода - *InitRMenu* и *InitEMenu*, которые инициализируют экземпляры *MenuBar* соответствующим образом: создается русско- либо англоязычное меню (метод начальной установки меню *InitMenuBar* вызывает один из этих методов). Затем в

методе, вызываемом при получении команды *cmSwitchMenu* (переключить меню), мы удаляем текущее меню, вызывая его деструктор, определяем текущее состояние меню (значение переменной *cMenu*) и вызываем либо метод *InitRMenu*, либо метод *InitEMenu*. После того как новое меню создано одним из доступных методов, мы отображаем его с помощью процедуры *Insert*.

Строка состояния

То же самое можно сделать и со строкой состояния, создаваемой с помощью метода *InitStatusLine*. Прежде всего, необходимо создать два метода, назовем их *InitRStatus* и *InitEStatus*. Переключение можно выполнять совместно с переключением меню.

В приведенной ниже программе показано, как осуществляется динамическое переключение строк состояния.

```
{-----
DYNASTAT.PAS. Динамически переключаемая строка
состояния
-----}
```

```
Uses

Objects,      {используем TRect}
Menus,        {используем группу объектов поддержки меню}
Drivers,      {используем обработчик событий}
Views,
App;          {используем объект TApplication}

Const
  cmSwitch     = 200; {команда переключения}

Type
  TMyApp      = Object(TApplication)

  cStat       : Boolean; {тип строки состояния}

  constructor Init;
  procedure InitStat;
  procedure InitEStat;
  procedure SwitchStatus;
  procedure InitStatusLine; virtual;
  procedure HandleEvent(var Event : TEvent); virtual;
End;

Constructor TMyApp.Init;
Begin
  TApplication.Init;
  cStat := False;
End;
```

```

Procedure TMyApp.InitRStat;
Var
  R : TRect;
Begin
  GetExtent(R);
  R.A.Y := R.B.Y - 1;
  StatusLine := New(PStatusLine, Init(R,
    NewStatusDef(0, $FFFF,
      NewStatusKey('~Alt-X~ Выход', kbAltX, cmQuit,
        NewStatusKey('~Alt-S~ Стереть', kbAltS, cmSwitch,
          nil)),
      nil)
    ));
End;

```

```

Procedure TMyApp.InitEStat;
Var
  R : TRect;
Begin
  GetExtent(R);
  R.A.Y := R.B.Y - 1;
  StatusLine := New(PStatusLine, Init(R,
    NewStatusDef(0, $FFFF,
      NewStatusKey('~Alt-X~ Exit', kbAltX, cmQuit,
        NewStatusKey('~Alt-S~ Status', kbAltS, cmSwitch,
          nil)),
      nil)
    ));
End;

```

```

Procedure TMyApp.SwitchStatus;
Begin
  {удалить текущую строку состояния}
  Delete(StatusLine);
  Dispose(StatusLine, Done);

  if cStat Then InitRStat else InitEStat;

  cStat := Not(cStat);

  {отобразить строку состояния}
  Insert(StatusLine);
End;

```

```

Procedure TMyApp.InitStatusLine;
Begin
  InitEStat;
End;

```

```

Procedure TMyApp.HandleEvent;
Begin
  TApplication.HandleEvent(Event);
  if Event.What = evCommand then
  Begin
    case Event.Command of
      cmSwitch : SwitchStatus;
    end;
  End;
  ClearEvent(Event);

```

```

End;

Var
  MyApp : TMyApp;

Begin

  MyApp.Init;
  MyApp.Run;
  MyApp.Done;

End.

```

В завершение рассмотрения различных объектов, реализованных в модуле APP, посмотрим, что еще полезного содержится в этом модуле.

Модуль APP

Стандартные меню и строки состояния

Модуль App содержит 4 новые функции, которые могут быть использованы для создания "стандартных" строк состояния и элементов меню. В приведенной ниже таблице показаны определяемые этими функциями команды и элементы меню.

Строка состояния: Функция StdStatusKeys

Клавиша	Команда
Alt-X	cmQuit
F10	cmMenu
Alt-F3	cmClose
F5	cmZoom
Ctrl-F5	cmResize
F6	cmNext

Меню File: Функция StdFileMenuItems

Элемент меню	Команда	Клавиша
New	cmNew	
Open	cmOpen	F3
Save	cmSave	F2
Save as	cmSaveAs	
Save all	cmSaveAll	

Change dir	cmChageDir	
DOS shell	cmDosShell	
Exit	cmQuit	Alt-X

Меню Edit: Функция StdEditMenuItems

Элемент меню	Команда	Клавиша
Undo	cmUndo	Alt-BackSpace
Cut	cmCut	Shift-Del
Copy	cmCopy	Ctrl-Ins
Paste	cmPaste	Shift-Ins
Clear	cmClear	Ctrl-Del

Меню Window: Функция StdWindowMenuItems

Элемент меню	Команда	Клавиша
Tile	cmTile	
Cascade	cmCascade	
Close all	cmCloseAll	
Size/Move	cmResize	Ctrl-F5
Zoom	cmZoom	F5
Next	cmNext	F6
Previous	cmPrev	Shift-F6
Close	cmClose	Alt-F3

Ниже показано использование функций, определяющих "стандартные" меню и элементы строки состояния:

{-----
STDMENU.PAS: Пример использования предопределенных меню и строки состояния
 -----}

```

uses App, Objects, Drivers, Menus, Views;

Const
  cmWindow = 100;
Type
  TMyApp = Object(TApplication)
    procedure InitStatusLine;          virtual;
    procedure InitMenuBar;            virtual;
  End;

Procedure TMyApp.InitStatusLine;
Var R : TRect;
Begin
  GetExtent(R); R.A.Y := R.B.Y - 1;
  New(StatusLine, Init(R,
    NewStatusDef($0,$FFFF,
      NewStatusKey('Alt-S Shell', kbAltS, cmDosShell,
```

```

NewStatusKey('~Alt-W~ Window', kbAltW,
                                     cmWindow,
StdStatusKeys(nil))),    {Предопределенные команды}
Nil));
End;

Procedure TMyApp.InitMenuBar;
Var R : TRect;
Begin
  GetExtent(R); R.B.Y := R.A.Y + 1;
  {Создать меню на основе предопределенных подменю}
  MenuBar := New(PMenuBar, Init(R, NewMenu(
    NewSubMenu('~F~ile', hcNoContext, NewMenu(
      StdFileMenuItems(nil)),
    NewSubMenu('~E~dit', hcNoContext, NewMenu(
      StdEditMenuItems(nil)),
    NewSubMenu('~W~indow', hcNoContext, NewMenu(
      StdWindowMenuItems(nil)),
    nil))));
End;

Var MyApp: TMyApp;

Begin
  MyApp.Init;
  MyApp.Run;
  MyApp.Done;
End.

```

Примечание: имеется возможность расширения стандартных меню. Для этого необходимо вставить элементы типа *PMenuitem* вместо параметра *Nil* соответствующей функции.

В модуле *APP* определен ряд стандартных команд, а также некоторые контексты справочной системы:

Команда	Знач.	Контекст	Знач.	Источник
cmNew	30	hcNew	\$FF01	File New
cmOpen	31	hcOpen	\$FF02	File Open
cmSave	32	hcSave	\$FF03	File Save
cmSaveAs	33	hcSaveAs	\$FF04	File Save As
cmSaveAll	34	hcSaveAll	\$FF05	File Save All
cmChangeDir	35	hcChangeDir	\$FF06	File Change Dir
cmDosShell	36	hcDosShell	\$FF07	File DOS Shell
cmCloseAll	37	hcCloseAll	\$FF22	File Close All

Отметим, что контексты справочной системы в диапазоне *\$FF00-\$FFFF* зарезервированы фирмой Borland для собственных нужд.

Следующие константы задают контексты справочной системы для ряда стандартных команд, определенных в других модулях.

Контекст	Значение	Источник
hcUndo	\$FF10	Edit Undo
hcCut	\$FF11	Edit Cut
hcCopy	\$FF12	Edit Copy
hcPaste	\$FF13	Edit Paste
hcClear	\$FF14	Edit Clear
hcTile	\$FF20	Window Tile
hcCascade	\$FF21	Window Cascade
hcResize	\$FF23	Window Size/Move
hcZoom	\$FF24	Window Zoom
hcNext	\$FF25	Window Next
hcPrev	\$FF26	Window Previous
hcClose	\$FF27	Window Close

Заключение

В модуле *APP* содержатся объекты, без которых не может обойтись ни одно Turbo Vision-приложение. Основным объектом является объект *TApplication*, методы которого задают способы поведения создаваемой прикладной программы. Более абстрактный объект *TProgram* задает основные характеристики всех приложений, тогда как объекты *TDesktop*, *TBackground*, *TMenuBar* и *TStatusLine* позволяют не только определить внешний вид приложения, но и задать основные пользовательские команды, определить горизонтальное и вложенное меню, а также поддерживать примитивные средства контекстно-зависимой подсказки.

ГЛАВА 3. Окна и панели диалога

Окна и панели являются наиболее часто используемыми интерфейсными элементами Turbo Vision и предназначены для взаимодействия с пользователем путем ввода/вывода различной информации. Панели диалога являются специальной формой окон и рассматриваются в этой главе вместе с окнами.

Окна

Окна в Turbo Vision реализованы, как групповой объект. Предком объекта *TWindow* является объект *TGroup*. Обычно в эту группу входят следующие объекты: *TFrame*, который отвечает за отображение рамки окна, *TScroller*, который отвечает за отображение информации в окне и один или два объекта *TScrollBar*.

Окно создается на базе объекта *TWindow*. После того как экземпляр такого объекта создан:

```
NewWindow := New(PWindow, Init(R, 'Window Title', wnNoNumber));
```

можно изменить его характеристики, например, местоположение окна :

```
NewWindow^.Options := NewWindow^.Options OR ofCentered;
```

Хотя прямой доступ к полям данных объекта является нарушением инкапсуляции, в Turbo Vision это единственный способ изменения свойств объектов. После того как все свойства окна заданы, оно помещается в приложение:

```
Application^.InsertWindow(NewWindow);
```

Новый метод *InsertWindow* возвращает указатель на окно. Если же мы создали объект-наследник *TWindow*, скажем, *TSpecialWindow*, все его свойства могут быть заданы при инициализации. Тогда метод *InsertWindow* может быть использован следующим образом:

```
Application^.InsertWindow(New(PSpecialWindow, Init(R, 'Title', wnNoNumber)));
```

Пример создания окна и его отображения показан ниже.

```
/////////////////////////////////////////////////////////////////
//WINDOW: Пример использования объекта TWindow
/////////////////////////////////////////////////////////////////
uses App, StdDlg, Objects, Drivers, Menus, Views, Dialogs, MsgBox;
Const
  cmNewWindow = 3000;
Type
  TDemoApp = Object(TApplication)
    procedure InitStatusLine;           virtual;
    procedure HandleEvent(var Event : TEvent);  virtual;
    procedure NewWindow;
  End;

Var
  W : PWindow;
Procedure TDemoApp.InitStatusLine;
Var
  R : TRect;
Begin
  GetExtent(R);
  R.A.Y := R.B.Y-1;
  StatusLine := New(PStatusLine, Init(R,
    NewStatusDef(0, $FFFF,
      NewStatusKey('~Alt-X~ Exit', kbAltX, cmQuit,
        NewStatusKey('~Alt-W~ ', kbAltW, cmNewWindow,
          nil)),
      nil)
  ));
End;
Procedure TDemoApp.HandleEvent;
Begin
  Inherited HandleEvent(Event);
  If Event.What = evCommand Then
    Begin
      If Event.Command = cmNewWindow Then NewWindow;
      ClearEvent(Event);
    End
  End;
End;
Procedure TDemoApp.NewWindow;
Var
  R : TRect;
Begin
  R.Assign(20, 5, 60, 15);
  W := New(PWindow, Init(R, '', wnNoNumber));
  InsertWindow(W);
End;

Var
  DemoApp : TDemoApp;
Begin
  DemoApp.Init;
  DemoApp.Run;
  DemoApp.Done;
End.
```

По умолчанию созданное окно использует палитру *wpBlueWindow*. Для изменения палитры необходимо изменить значение поля *Palette* объекта *TWindow*. В приведенном ниже примере показано, как это сделать.

```

/////////////////////////////////////////////////////////////////
WINDEMO: Установка цветов окна
/////////////////////////////////////////////////////////////////
uses App, StdDlg, Objects, Drivers, Menus, Views, Dialogs,
    MsgBox;
Const
    cmNewWindow = 3000;
Type
    TDemoApp = Object(TApplication)
        procedure InitStatusLine;           virtual;
        procedure HandleEvent(var Event : TEvent); virtual;
        procedure NewWindow;
    End;

Procedure TDemoApp.InitStatusLine;
Var
    R : TRect;
Begin
    GetExtent(R);
    R.A.Y := R.B.Y-1;
    StatusLine := New(PStatusLine, Init(R,
        NewStatusDef(0, $FFFF,
            NewStatusKey('~Alt-X~ Exit', kbAltX, cmQuit,
            NewStatusKey('~Alt-W~ ', kbAltW, cmNewWindow,
            nil)),
        nil)
    ));
End;
Procedure TDemoApp.HandleEvent;
Begin
    Inherited HandleEvent(Event);
    If Event.What = evCommand Then
        Begin
            If Event.Command = cmNewWindow Then NewWindow;
            ClearEvent(Event);
        End
    End;
Procedure TDemoApp.NewWindow;
Var
    R : TRect;
    W : PWindow;
Begin
    R.Assign(20,5,60,15);
    W := New(PWindow, Init(R, "wnNoNumber));
    {..
        Изменить палитру окна: возможны значения:
        wpBlueWindow, wpCyanWindow, wpGrayWindow
    ..}
    W.Palette := wpCyanWindow;

    InsertWindow(W);
End;

```

```

Var
  DemoApp : TDemoApp;
Begin
  DemoApp.Init;
  DemoApp.Run;
  DemoApp.Done;
End.

```

Установка свойств окна

Поле *Flags* позволяет установить ряд характеристик окна. По умолчанию устанавливается следующее значение этого поля:

```
Flags := wfMove + wfGrow + wfClose + wfZoom
```

Это значение поля *Flags* позволяет перемещать окно, изменять его размер, закрывать окно и максимизировать его. Возможно использование следующих значений поля *Flags*:

Флаг	Назначение
wfMove	Возможно перемещение окна
wfGrow	Возможно изменение размеров окна
wfClose	Возможно закрытие окна (кнопка Close)
wfZoom	Возможна максимизация окна (кнопка Zoom)

Поле *State* задает еще одну группу характеристик. По умолчанию значение этого поля равно *sfShadow*. Это указывает на то, что окно должно отбрасывать тень.

Отображение информации в окне

Объект *TWindow* сам по себе не отвечает за содержимое окна. Метод *Draw* этого объекта даже не переопределяется. Метод же *TGroup.Draw* просто вызывает перерисовку объектов, включенных в группу (метод *DrawSubViews*). Если мы проследим цепочку объектов, то увидим, что метод *TWindow.Draw* выполняет следующие действия:

```

MoveChar(B, ' ', GetColor(1), Size.X);
WriteLine(0, 0, Size.X, Size.Y, B);

```

т.е. просто заполняет область окна символами "пробел". Таким образом, для того, чтобы отобразить в окне что-нибудь осмысленное, нам необходимо либо переопределить

метод *TWindow.Draw*, либо поместить внутрь окна специальный отображаемый объект, который бы отвечал за содержимое окна. Обычно поступают следующим образом: создается объект - наследник объекта *TView*, метод *Draw* которого выполняет все необходимые действия по заполнению содержимого окна. Назовем такой объект *TInsider*.

```
{-----
WINDOW.PAS: Объект TInsider, отвечающий за заполнение окна
-----}
```

```
uses Dos, Objects, App, Dialogs, Views, Menus, Drivers, MsgBox;
```

```
Const
  cmWindow = 100;
Var
  R      : TRect;
Type
  PInsider = ^TInsider;
  TInsider = Object(TView)
    constructor Init(var Bounds : TRect);
    procedure Draw;                virtual;
  End;

  PMyWindow = ^TMyWindow;
  TMyWindow = Object(TWindow)
    constructor Init(var Bounds : TRect; WindowTitle :
      TTitleStr; WindowNo : Integer);
  End;

  TDAppl = Object(TApplication)
    Window : PMyWindow;
    procedure InitStatusLine;        virtual;
    procedure HandleEvent(var Event : TEvent);  virtual;
    procedure NewWindow;
  End;

  Constructor TInsider.Init;
  Begin
    Inherited Init(Bounds);
    GrowMode := gfGrowHiX + gfGrowHiY;
  End;
  Procedure TInsider.Draw;
  Begin
    Inherited Draw;
  End;

  Constructor TMyWindow.Init;
  Begin
    Inherited Init(Bounds, WindowTitle, wnNoNumber);
    GetClipRect(Bounds);
    Bounds.Grow(-1, -1);
    Insert(New(PInsider, Init(Bounds)));
  End;

  Procedure TDAppl.InitStatusLine;
  Begin
```

```

GetExtent(R);
R.A.Y := R.B.Y - 1;
StatusLine := New(PStatusLine, Init(R,
  NewStatusDef(0, $FFFF,
    NewStatusKey('~Alt-X~ Exit', kbAltX, cmQuit,
      NewStatusKey('~Alt-W~ Window', kbAltW, cmWindow,
        Nil)),
    Nil)
  ));
End;
Procedure TDAApp.NewWindow;
Begin
  R.Assign(0,0,40,15);
  Window := New(PMyWindow, Init(R, "wnNoNumber));
  Window.Options := Window.Options OR ofCentered;
  Application.InsertWindow(Window);
End;

Procedure TDAApp.HandleEvent;
Begin
  inherited HandleEvent(Event);
  if Event.What = evCommand Then
    Begin
      Case Event.Command of
        cmWindow : NewWindow;
      else Exit;
      End;
    End;
  ClearEvent(Event);
End;

Var
  DApp : TDAApp;

Begin
  DApp.Init;
  DApp.Run;
  DApp.Done;
End.

```

Реализованный выше объект *TInsider* содержит метод *Draw*, который в данной версии просто вызывает метод *Draw* своего предка. Для отображения какой-либо полезной информации, например, для отображения содержимого текстового файла необходимо расширить функциональность метода *Draw*. Самым простым способом является построчное считывание содержимого файла в какой-либо буфер или коллекцию строк, а затем отображение этих строк с помощью процедуры *WriteLine*.

Приведенный выше способ отображения информации в окне имеет существенное ограничение. Если число строк в файле больше вертикального размера окна, то мы не сможем рассмотреть часть из них. Решением такой проблемы является использование полос прокрутки - объектов типа *TScrollBar*. В Turbo Vision существует

специальный объект, предназначенный для отображения информации с возможностью ее прокрутки - *TScroller*. В его конструкторе помимо размера указываются два дополнительных параметра: указатели на горизонтальную и вертикальную строку прокрутки. Таким образом, если мы создадим объект *TInsider* как наследник объекта *TScroller*, то сможем выполнять скроллинг информации в окне. При использовании полос прокрутки необходимо помнить, что текст должен отображаться в зависимости от позиции бегунка полосы прокрутки - поле *Delta*.

Модальные окна

Возможно создание модальных окон. Для этого вместо метода *InsertWindow* мы должны использовать метод *ExecuteDialog*:

```
NewWindow := New(PWindow, Init(R,'Modal Window', wnNoNumber));
ExecuteDialog(PDialog(NewWindow),Nil);
```

Для завершения работы модального окна необходимо переопределить метод *HandleEvent* и в ответ на одно из событий вызвать метод *EndModal*.

Пример реализации модального окна приведен ниже.

```
{//////////////////////////////////////
MODALWND.PAS: Пример создания модального окна
//////////////////////////////////////}
uses Objects, App, Drivers, Views, Dialogs, Menus;
Const
  cmMWnd    = 100;
Type
  PNewWindow = ^TNewWindow;
  TNewWindow = Object(TWindow)
    procedure HandleEvent(var Event : TEvent);   virtual;
  End;

  TDemoApp = Object(TApplication)
    procedure InitStatusLine;                     virtual;
    procedure HandleEvent(var Event : TEvent);   virtual;
    procedure NewWindow;
  End;

Procedure TNewWindow.HandleEvent;
Begin
  Inherited HandleEvent(Event);
  if Event.What = evCommand then
  begin
    case Event.Command of
      cmCancel : EndModal(cmCancel);
    else
```

```

    exit;
end;
ClearEvent(Event);
end;

End;

Procedure TDemoApp.InitStatusLine;
Var
  R : TRect;
Begin
  GetExtent(R);
  R.A.Y := R.B.Y - 1;
  StatusLine := New(PStatusLine, Init(R,
    NewStatusDef(0, $FFFF,
      NewStatusKey('~Alt-X~ Exit', kbAltX, cmQuit,
        NewStatusKey('~Alt-M~ MWnd', kbAltM, cmMWnd,
          Nil)),
      Nil)
    ));
End;
Procedure TDemoApp.HandleEvent;
Begin
  Inherited HandleEvent(Event);
  if Event.What = evCommand then
  begin
    case Event.Command of
      cmMWnd : NewWindow;
    else
      exit;
    end;
    ClearEvent(Event);
  end;
End;
Procedure TDemoApp.NewWindow;
Var
  MWindow : PNewWindow;
  R : TRect;
Begin
  R.Assign(20,5,60,15);
  MWindow := New(PNewWindow, Init(R, 'Modal Window',
    wnNoNumber));
  ExecuteDialog(PDialog(MWindow), nil);
End;

Var
  DemoApp : TDemoApp;
Begin
  DemoApp.Init;
  DemoApp.Run;
  DemoApp.Done;
End.

```

Примечание: завершение модального состояния окна происходит по активизации кнопки закрытия окна.

Меню в окне

Те, кто пользовался средой Microsoft Windows, наверняка заметил, что окна в этой среде имеют меню, которое может вызываться специальной кнопкой. Окнам в Turbo Vision также можно придать такую функциональность. Для этого необходимо переопределить объект *TFrame*, который используется для реализации рамки окна. В приведенном ниже примере показано, как подсоединить меню к кнопке закрытия окна.

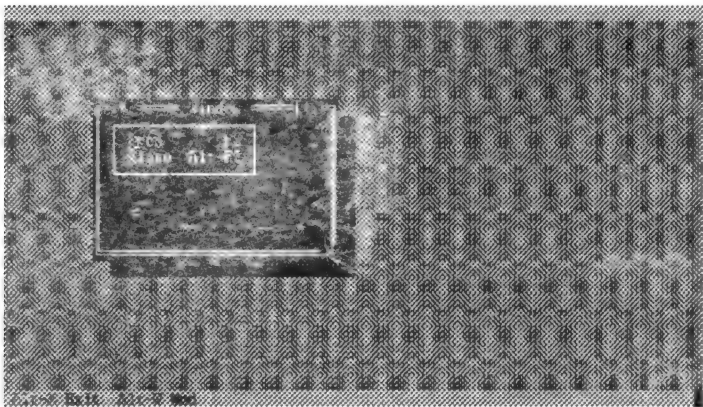


Рис.3.1. Меню в окне.

```
/////////////////////////////////////////////////////////////////
WIN_MENU: Окно, содержащее меню
/////////////////////////////////////////////////////////////////
Uses Objects, Views, App, Drivers, MsgBox, Menus;

Const
  cmNewWin = 1000;
Type
  PMyFrame = ^TMyFrame;
  TMyFrame = Object(TFrame)
    procedure HandleEvent(var Event : TEvent);virtual;
  End;

  PMyWindow = ^TMyWindow;
  TMyWindow = Object(TWindow)
    procedure InitFrame;virtual;
  End;

  TMyApp = Object(TApplication)
    procedure InitStatusLine; virtual;
    procedure HandleEvent(var Event : TEvent);virtual;
    procedure NewWindow;
```

```

End;

Procedure TMyFrame.HandleEvent;
Var
  Mouse : TPoint;
  R : TRect;
  MP: PMenuPopup;
Begin
  TView.HandleEvent(Event);
  if (Event.What = evMouseDown) and (State and sfActive <> 0) Then
  Begin
    MakeLocal(Event.Where, Mouse);
    If Mouse.Y = 0 Then
      Begin
        if (PWindow(Owner) ^ .Flags and wfClose <> 0) and
          (Mouse.X >= 2) and (Mouse.X <= 4) Then
          Begin
            GetExtent(R);
            Inc(R.A.X);Inc(R.A.Y);
            MP := New(PMenuPopup,Init(R,
              NewMenu(
                NewItem('O~pen','F3',kbF3,cmCancel,hcNoContext,
                  NewItem('C~lose','Alt-F3',kbAltF3,cmClose,
                    hcNoContext,
                    Nil)))
              ));
            Owner ^ .Insert(MP);
            ClearEvent(Event);
          End;
        End;
      End;
    Inherited HandleEvent(Event);
  End;

Procedure TMyWindow.InitFrame;
Var
  R : TRect;
Begin
  GetExtent(R);
  Frame := New(PMyFrame, Init(R));
End;

Procedure TMyApp.InitStatusLine;
Var
  R : TRect;
Begin
  GetExtent(R);
  R.A.Y := R.B.Y - 1;
  StatusLine := New(PStatusLine,Init(R,
    NewStatusDef(0, $FFFF,
      NewStatusKey('Alt-X~Exit', kbAltX, cmQuit,
        NewStatusKey('Alt-W~Wnd ', kbAltW, cmNewWin,
          Nil)),
      ));
End;

Procedure TMyApp.HandleEvent;
Begin
  Inherited HandleEvent(Event);

```

```

If Event.What = evCommand Then
Begin
  Case Event.Command of
    cmNewWin : NewWindow;
  else
    Exit;
  End;
  ClearEvent(Event);
End;
End;

Procedure TMyApp.NewWindow;
Var
  Window : PMyWindow;
  R:TRect;
Begin
  R.Assign(10,5,38,15);
  New(Window,Init(R,'Window',wnNoNumber));
  InsertWindow(Window);
End;

Var
  MyApp : TMyApp;

Begin
  MyApp.Init;
  MyApp.Run;
  MyApp.Done;
End.

```

При нажатии кнопки закрытия окна вы получите меню, содержащее два элемента: *Open* и *Close*. По команде *Close* окно закрывается. Для того, чтобы придать окну свойства окна в среде Microsoft Windows, необходимо реализовать как минимум четыре команды:

Элемент меню	Команда
Zoom	cmZoom
Move/Size	cmResize
Close	cmClose
Next Window	cmNext

Примечание: объект *TWindow* имеет поле *Frame* типа *PFrame*, которое ассоциирует окно с рамкой. Рамка может содержать кнопку закрытия окна, кнопку распахивания окна, реализует возможность перемещения и изменения размеров окна.

Таким образом, переопределив метод *TWindow.InitFrame*, мы можем создать собственную рамку.

Нажатие кнопки закрытия окна реализуется следующим образом: если произошло событие от "мыши", то

проверяется Y-координата. Для этого, координаты курсора "мыши" преобразуются в локальные для данного объекта:

```
MakeLocal(Event.Where, Mouse);
```

Если Y-координата равна 0, то проверяется наличие кнопки закрытия окна (*wfClose*):

```
if (PWindow(Owner)~.Flags AND wfClose <> 0
```

Отметим, что кнопка закрытия реализована не как объект типа *TButton*. Затем, если "кнопка" закрытия окна существует, то выполняется проверка X-координаты курсора "мыши". Если он находится в диапазоне от 2 до 4, то считается, что "кнопка" закрытия окна нажата. В этом случае посылается сообщение:

```
Event.What := evCommand  
Event.Command := cmClose;  
(* Послать сообщение владельцу рамки*)  
Event.InfoPtr := Owner;  
PutEvent(Event);  
ClearEvent(Event);
```

Использование объекта *TFrame*

Еще одним примером использования объекта *TFrame* может быть следующий: предположим, у нас есть группа отображаемых объектов, которые, по каким-либо причинам должны выделяться среди остальных объектов в панели диалога. Такие объекты можно ограничить рамкой. Казалось бы, что для этих целей подходит объект *TGroup* - необходимо указать значение поля *Options* как *Options OR ofFramed*, и мы получим необходимый объект. Но такое решение не подходит, так как объект *TGroup* не отвечает за отображение всей области экрана, которую он занимает. Если мы будем использовать объект *TFrame*, то мы сможем получить желаемый эффект. Если использовать объект *TFrame* как есть, то мы столкнемся с двумя побочными эффектами. Во первых, объект *TFrame* копирует свойства объекта-владельца. Например, если у панели диалога, в которую помещена, рамка присутствует кнопка закрытия, у рамки также будет присутствовать такая кнопка. Выбор рамки с помощью мыши приведет к выбору панели диалога, метод *HandleEvent* объекта *TFrame* передает основные события своему владельцу. Таким образом, нам как минимум необходимо переопределить метод

HandleEvent. Отказаться от отображения кнопок можно переопределив метод *Draw*, или отменив наличие кнопки закрытия у самой панели диалога. Если мы хотим, чтобы рамка не откликалась ни на какие события, нам необходимо переопределить метод *HandleEvent* таким образом, чтобы он вообще не обрабатывал их. Пример использования объекта *TFrame* в качестве рамки, объединяющей группу объектов, приведен ниже.



Рис.3.2. Рамка вокруг объектов

{-----
FRAME.PAS: Пример использования объекта TFrame
-----}

```
uses Dos, Objects, App, Dialogs, Views, Menus, Drivers, MsgBox;
Const
  cmNewDlg = 100;
Var
  R      : TRect;
Type
  TDAppl = Object(TApplication)
    Dialog : PDialog;
    procedure InitStatusLine;          virtual;
    procedure HandleEvent(var Event : TEvent);  virtual;
    procedure NewDlg;
  End;
{Static Frame}
PSFrame = ^TSFrame;
TSFrame = Object(TFrame)
  procedure HandleEvent(var Event : TEvent);  virtual;
End;

Procedure TSFrame.HandleEvent;
Begin
End;

Procedure TDAppl.InitStatusLine;
```

```

Begin
  GetExtent(R);
  R.A.Y := R.B.Y - 1;
  StatusLine := New(PStatusLine, Init(R,
    NewStatusDef(0, $FFFF,
      NewStatusKey('~Alt-X~ Exit', kbAltX, cmQuit,
        NewStatusKey('~Alt-D~ Dialog', kbAltD, cmNewDlg,
          Nil)),
      Nil)
  ));
End;
Procedure TDAApp.NewDlg;
Var
  OK      : PButton;
  F       : PSFrame;
Begin
  R.Assign(0,0,60,20);
  Dialog := New(PDialog, Init(R, ""));
  With Dialog do
    Begin
      Options := Options OR ofCentered;
      Flags := Flags AND NOT ofClose;
      Palette := dpCyanDialog;
      R.Assign(10,2,50,18);
      F := New(PSFrame, Init(R));
      Insert(F);
    End;
  Application.ExecView(Dialog);
End;

Procedure TDAApp.HandleEvent;
Begin
  inherited HandleEvent(Event);
  if Event.What = evCommand Then
    Begin
      Case Event.Command of
        cmNewDlg : NewDlg;
        else Exit;
      End;
    End;
  ClearEvent(Event);
End;

Var
  DApp : TDAApp;

Begin
  DApp.Init;
  DApp.Run;
  DApp.Done;
End.

```

Панели диалога

Панели диалога используются совместно с другими элементами управления. Эти элементы - кнопки, списки,

поля ввода - помещаются в панель после ее создания с помощью метода Insert:

```
Procedure TDemoApp.NewDialogBox;
Var
  Dialog : PDialog;
  R      : TRect;
Begin
  R.Assign(20,5,60,15);
  Dialog := New(PDialog, Init(R,'New Dialog'));
  With Dialog do
  begin
    R.Assign(15,10,25,12);
    Insert(New(PButton,Init(R,'O~k',cmOk,bfDefault)));
    R.Assign(30,10,40,12);
    Insert(New(PButton,Init(R,'Cancel',cmCancel,bfNormal)));
  end;
  Desktop.ExecView(Dialog);
End;
```

Стандартные панели диалога

В состав Turbo Vision входит ряд стандартных панелей диалога. Наиболее часто используемой является панель сообщений. Обычно, в программе панель сообщений используется для уведомления пользователя или получения подтверждения перед выполнением определенных действий:

```
Reply := MessageBox('Delete file ?',Nil, mfOkButton);
```

Панель сообщения, отображаемая с помощью функции *MessageBox* имеет фиксированный размер: 40 символов x 9 строк. При необходимости отображения более длинного сообщения можно использовать функцию *MessageBoxRect*. Функция *MessageBox* - это упрощенный вид функции *MessageBoxRect*.

Изменения в Turbo Vision 2.0

Единственное изменение, произошедшее в этом модуле, - введение флага *mfInsertInApp*. Этот флаг позволяет указать тот объект, который является владельцем панели сообщения. Флаг *mfInsertInApp* используется в комбинации с другими флагами:

```
MessageBox('MsgBox Demo', Nil, mfInformation OR mfOkButton OR
mfInsertInApp);
```

Если этот флаг не установлен, владельцем панели сообщения является объект *TDesktop*:

```
MessageBox := Desktop^.ExecView(MBX)
```

При установленном флаге владельцем панели сообщения является объект *TApplication*:

```
MessageBox := Application^.ExecView(MBX)
```

Функция InputBox

Эта функция, реализованная в модуле *MsgBox*, используется для отображения панели диалога, состоящей из заголовка и строки ввода, а также кнопок ОК и Cancel. Обычно эта функция используется для ввода какого-либо текста, например, имени файла. Использование функции *InputBox* показано ниже.

```
{//////////////////////////////////////  
INPUTBOX.PAS: Пример использования функции InputBox  
//////////////////////////////////////}  
uses Objects, App, Drivers, Views, Dialogs, Menus, MsgBox;  
Const  
  cmFileName   = 1000;  
Type  
  TDemoApp = Object(TApplication)  
    sFileName : String;  
    procedure InitStatusLine;          virtual;  
    procedure HandleEvent(var Event : TEvent);  virtual;  
  End;  
  
Procedure TDemoApp.InitStatusLine;  
Var  
  R : TRect;  
Begin  
  GetExtent(R);  
  R.A.Y := R.B.Y - 1;  
  StatusLine := New(PStatusLine, Init(R,  
    NewStatusDef(0, $FFFF,  
      NewStatusKey('~Alt-X~ Exit', kbAltX, cmQuit,  
        NewStatusKey('~Alt-F~ Input', kbAltF, cmFileName,  
          Nil)),  
    Nil)  
  ));  
End;  
Procedure TDemoApp.HandleEvent;  
Begin  
  Inherited HandleEvent(Event);  
  if Event.What = evCommand then  
    begin  
      case Event.Command of  
        cmFileName : InputBox('Введите имя файла', 'Имя  
                               файла : ', sFileName, 30);
```

```

else
  exit;
end;
ClearEvent(Event);
end;
End;

Var
  DemoApp : TDemoApp;
Begin
  DemoApp.Init;
  DemoApp.Run;
  DemoApp.Done;
End.

```

Функция InputBoxRect

Эта функция выполняет действия, аналогичные функции InputBox, но имеет дополнительный параметр, позволяющий задать ее координаты и размер. Функция InputBox - более упрощенный вид функции InputBoxRect: в ней создается панель диалога размером 60 символов x 8 строк, которая затем помещается в центр экрана.

Объект TFileDialog

Для загрузки и сохранения файлов может использоваться панель диалога *TFileDialog*:

```

Procedure TDemoApp.OpenFile;
Var
  FileDialog : PFileDialog;
Const
  FDOptions : Word = fdOKButton + fdOpenButton;
Begin
  TheFile := '.EXE';
  FileDialog :=
    New(PFileDialog, Init(TheFile, 'Open file', '~File
    name',
    FDOptions, 0));
  If ExecuteDialog(FileDialog, @TheFile) <> cmCancel Then
  Begin
    CheckFile;
  End;
End;

```

В приведенном выше примере имя выбранного файла возвращается в глобальной переменной *TheFile*.

Для изменения текущего каталога может использоваться панель диалога *TChDirDialog*. Обе панели диалога

реализованы в модуле *StdDlg*. Ниже приводится пример использования панели диалога *TChDirDialog*.

```

{//////////////////////////////////////////////////////////////////
CH_DLГ.PAS:   Пример   использования   панели   диалога
TChDirDialog
//////////////////////////////////////////////////////////////////}

uses App, StdDlg, Objects, Drivers, Menus, Views;
Const
  cmChDir = 3000;
Type
  TDemoApp = Object(TApplication)
    procedure InitStatusLine;           virtual;
    procedure HandleEvent(var Event : TEvent);  virtual;
    procedure ChDir;
  End;

Var
  SaveDir : String;

Procedure TDemoApp.InitStatusLine;
Var
  R : TRect;
Begin
  GetExtent(R);
  R.A.Y := R.B.Y-1;
  StatusLine := New(PStatusLine, Init(R,
    NewStatusDef(0, $FFFF,
      NewStatusKey('~Alt-X~ Exit', kbAltX, cmQuit,
        NewStatusKey('~Alt-C~ ChDir', kbAltC, cmChDir,
          nil)),
      nil)
  ));
End;
Procedure TDemoApp.HandleEvent;
Begin
  Inherited HandleEvent(Event);
  If Event.What = evCommand Then
    Begin
      If Event.Command = cmChDir Then ChDir;
      ClearEvent(Event);
    End
  End;
Procedure TDemoApp.ChDir;
Var
  D: PChDirDialog;
Begin
  D := New (pChDirDialog, Init (cdNormal + cdHelpButton,
    101));
  If ValidView (D) <> nil Then Begin
    DeskTop^.ExecView (D);
    Dispose (D, Done);
    GetDir (0, SaveDir);
  End;
End;

Var
  DemoApp : TDemoApp;

```

```

Begin
  DemoApp.Init;
  DemoApp.Run;
  DemoApp.Done;
End.

```

Панель диалога *TChDirDialog* вызывается по команде *cmChDir (Alt-C)* и результат (текущий каталог) сохраняется в глобальной переменной *SaveDir*.

Расширение функциональности

Расширение функциональности стандартных панелей диалога не вызывает больших затруднений. Например, чтобы добавить кнопку *Info*, которая будет возвращать информацию о текущем каталоге, необходимо переопределить метод *TChDirDialog.HandleEvent* и ввести обработку еще одного сообщения (например, *cmInfo*), а также после инициализации панели диалога, но перед ее отображением, вставить необходимый элемент управления. В приведенном ниже примере показано, как добавить новую кнопку к панели диалога *TChDirDialog* и как переопределить метод *HandleEvent*.

////////////////////////////////////
 CH_DLG1.PAS: Пример расширения функций панели диалога
 TChDirDialog

////////////////////////////////////}

```

uses App, StdDlg, Objects, Drivers, Menus, Views, Dialogs, MsgBox;

```

```

Const

```

```

  cmChDir = 3000;

```

```

  cmInfo = 3001;

```

```

Type

```

```

  TDemoApp = Object(TApplication)

```

```

    procedure InitStatusLine; virtual;

```

```

    procedure HandleEvent(var Event : TEvent); virtual;

```

```

    procedure ChDir;

```

```

  End;

```

```

  PChDir = ^TChDir;

```

```

  TChDir = Object(TChDirDialog)

```

```

    procedure HandleEvent(var Event : TEvent); virtual;

```

```

  End;

```

```

Var

```

```

  SaveDir : String;

```

```

Procedure TChDir.HandleEvent;

```

```

Begin

```

```

  Inherited HandleEvent(Event);

```

```

  If Event.What = evCommand Then

```

```

    Begin

```

```

      If Event.Command = cmInfo Then

```

```

    Begin
    {** Информация отображается здесь **}
    MessageBox{#3'Info',Nil,mfOkButton);
    End;
    ClearEvent(Event);
End
End;

Procedure TDemoApp.InitStatusLine;
Var
  R : TRect;
Begin
  GetExtent(R);
  R.A.Y := R.B.Y-1;
  StatusLine := New(PStatusLine,Init(R,
    NewStatusDef(0, $FFFF,
    NewStatusKey('~Alt-X~ Exit', kbAltX, cmQuit,
    NewStatusKey('~Alt-C~ ChDir',kbAltC, cmChDir,
    nil)),
    nil)
  ));
End;
Procedure TDemoApp.HandleEvent;
Begin
  Inherited HandleEvent(Event);
  If Event.What = evCommand Then
    Begin
      If Event.Command = cmChDir Then ChDir;
      ClearEvent(Event);
    End
End;
Procedure TDemoApp.ChDir;
Var
  D : PChDir;
  B : PButton;
  R : TRect;
Begin
  D := New (PChDir, Init (cdNormal, 101));

  {* Вставить кнопку Info *}
  R.Assign(35,3,45,5);
  B := New(PButton, Init(R, '~I~nfo', cmInfo, bfNormal));
  D.Insert(B);
  {* *}
  If ValidView (D) <> nil Then Begin
    DeskTop.ExecView (D);
    Dispose (D, Done);
    GetDir (0, SaveDir);
  End;
End;

Var
  DemoApp : TDemoApp;
Begin
  DemoApp.Init;
  DemoApp.Run;
  DemoApp.Done;
End.

```

Если требуется изменить размеры панели управления, например, для добавления новых интерфейсных элементов, то это может быть сделано следующим образом:

```
{* Изменить размер *}
D^.GetExtent(R);
R.B.Y := R.B.Y + 2;
D^.ChangeBounds(R);
```

D[^] - переменная типа *PDialog*. Метод *GetExtent* возвращает текущие координаты и размер отображаемого элемента. Затем, с помощью метода *ChangeBounds* устанавливаются новые координаты и размер панели диалога. Этот код необходимо поместить перед отображением панели диалога.

Элементы управления

Элементы управления, реализованные в Turbo Vision, используются совместно с панелями диалога. В составе панели диалога могут использоваться:

- обычные кнопки - объект *TButton*;
- кнопки с зависимой фиксацией - объект *TRadioButtons*;
- кнопки с независимой фиксацией - объект *TCheckBoxes*;
- списки - объект *TListBox*;
- строки ввода - объект *TInputLine*;
- протоколы - объект *THistory*;
- метка - объект *TLabel*;
- текст - объект *TStaticText*.

Кратко выделим основные моменты использования элементов управления. После того как объект создан (с помощью конструктора *Init*), возможно изменение его свойств поля объекта доступны для изменения:

```
NewButton := New(PButton, Init(R, 'O~k~', cmOk, bfDefault);
{* Изменение свойств *}
NewButton^.Options := NewButton^.Options OR ofCenterX;
{*}
Insert(NewButton);
```

Как отмечалось выше, для передачи данных более удобно реализовать специальную запись (*Record*), которая

может использоваться для задания начальных значений элементам управления и получения введенных данных.

При создании такой записи можно пользоваться приведенной ниже таблицей, в которой показаны размеры данных для каждого элемента управления.

Элемент управления	Размер данных
TButton	0
TCheckBoxes	2
TInputLine	MaxLen + 1
TLabel	0
TListBox	6
TParamText	ParamCount * 4
TRadioButtons	2
TScrollBar	0
TStaticText	0

Элементы управления и фокус

Необходимо отметить еще один момент использования элементов управления внутри панели диалога. Когда элементы управления помещаются в панель диалога с помощью метода *Insert*, каждый элемент управления становится "последним" в цепочке элементов (поле *Last* будет содержать указатель на этот объект). Таким образом, по умолчанию фокус будет иметь тот элемент, который был помещен в группу последним. Так, например, если заполнение панели диалога происходит следующим образом:

```
Procedure TDAppl.NewDlg;
Var
  OK      : PButton;
  Line1   : PInputLine;
  Line2   : PInputLine;
Begin
  R.Assign(0,0,40,15);
  Dialog := New(PDialog,Init(R,""));
  With Dialog do
    Begin
      Options := Options OR ofCentered;
      R.Assign(0,5,30,6);
      Line1 := New(PInputLine,Init(R,128));
      Line1.Options := Line1.Options OR ofCenterX;
      Insert(Line1);

      R.Assign(0,7,30,8);
      Line2 := New(PInputLine,Init(R,128));
```

```

Line2.Options := Line2.Options OR ofCenterX;
Insert(Line2);

R.Assign(0, 10, 10, 12);
OK := New(PButton, Init(R, 'O'k', cmOk, bfDefault));
OK.Options := OK.Options OR ofCenterX;
Insert(OK);
End;
Application.ExecView(Dialog);
End;

```

фокус будет иметь кнопка "OK", так как она помещена в панель диалога последней. Если же нам необходимо установить фокус на первую строку ввода, необходимо выполнить метод *Dialog.SelectNext* или метод *FocusNext*, реализованный в Turbo Vision 2.0, указав в качестве параметра *False*. Это приведет к тому, что будет выбран следующий элемент внутри панели диалога. Если же нужно установить фокус на какой-либо иной элемент, то необходимо вызывать метод *Select* этого элемента, например,

```
Line2.Select
```

Используя поле *TGroup.Last*, которое является указателем на объект типа *PView* и его поле *Next*, можно изменить порядок обхода цепочки элементов управления внутри группы. Так, первый элемент группы может быть получен следующим образом:

```
FirstView := Last.Next
```

и так далее. Это может быть полезно при создании интерактивных средств редактирования панелей диалога.

Объект *TButton*

Объект *TButton* предназначен для создания обычных кнопок, таких как кнопки, используемые для подтверждения или отмены определенных действий: Ok или Cancel. При нажатии кнопка посылает присвоенную ей команду. Кнопка может быть выбрана командной клавишей, клавишей Tab, нажатием клавиши Enter или манипулятором мышь. Обычно в большинстве приложений используется только конструктор *TButton.Init*. При задании координат и размеров кнопки необходимо помнить о том, что кнопка отображается как трехмерный объект: размер

тени должен быть включен в размер самой кнопки. Ниже показано использование объекта *TButton*.

```
{//////////////////////////////////////
BUTTON.PAS: Пример использования объекта TButton
//////////////////////////////////////}
```

```
uses Objects, Drivers, Views, Menus, Dialogs, App, MsgBox;
```

```
Const
  cmNewDialog = 101;
```

```
Type
  TMyApp = Object(TApplication)
    procedure HandleEvent(var Event: TEvent); virtual;
    procedure InitStatusLine; virtual;
    procedure NewDialog;
  End;
```

```
Procedure TMyApp.HandleEvent(var Event: TEvent);
Begin
  TApplication.HandleEvent(Event);
  if Event.What = evCommand then
    begin
      case Event.Command of
        cmNewDialog: NewDialog;
      else Exit;
      end;
      ClearEvent(Event);
    end;
End;
```

```
Procedure TMyApp.InitStatusLine;
var R: TRect;
Begin
  GetExtent(R);
  R.A.Y := R.B.Y - 1;
  StatusLine := New(PStatusLine, Init(R,
    NewStatusDef(0, $FFFF,
      NewStatusKey('~Alt-X~ Exit', kbAltX, cmQuit,
        NewStatusKey('~Alt-D~ Dialog', kbAltD, cmNewDialog,
          nil))),
    nil));
End;
```

```
Procedure TMyApp.NewDialog;
Var
  Dlg      : PDialog;
  R        : TRect;
  OkButton : PButton;
  CancelButton : PButton;
Begin
  R.Assign(20, 2, 60, 16);
  Dlg := New(PDialog, Init(R, 'Demo Dialog'));
  with Dlg do
    begin
      R.Assign(5, 10, 15, 12);
      OkButton := New(PButton, Init(R, '~O~k', cmOk,
        bfDefault));
```

```

Insert(OkButton);
R.Assign(25, 10, 35, 12);
CancelButton := New(PButton, Init(R, 'C~ancel',
cmCancel, bfNormal));
Insert(CancelButton);
SelectNext(False);
end;
DeskTop^.ExecView(Dlg);
End;

Var
MyApp: TMyApp;

Begin
MyApp.Init;
MyApp.Run;
MyApp.Done;
End.

```

В приведенном выше примере внутри панели диалога создаются две кнопки: *Ok* и *Cancel*, которые посылают команды *cmOk* и *cmCancel* соответственно. Кнопка *Ok* является кнопкой по умолчанию (*bfDefault*). Кнопка *Ok* может быть выбрана командной клавишей Alt-O, а кнопка *Cancel* - командной клавишей Alt-C.

В завершение рассмотрения стандартных свойств объекта *TButton*, посмотрим, как работает метод *HandleEvent*. Команда посылается владельцу кнопки следующим образом:

```

Event.What := evCommand;
Event.Command := Command; {Поле объекта TButton}
Event.InfoPtr := @Self;
PutEvent(Event);

```

Посылка команды реализована в методе *TButton.Press*.

Нестандартные кнопки

Казалось бы, что реализованные в Turbo Vision интерфейсные элементы охватывают все случаи жизни, но небольшой пример может показать, что это не так: попробуйте реализовать панель выбора дисководов с помощью стандартных кнопок: у вас не получится расположить кнопки в один ряд, так как каждая кнопка занимает как минимум 5 позиций. Решением этой проблемы может стать создание нестандартной кнопки. Реализуем такую кнопку (она будет плоской в нашем примере):

```

/////////////////////////////////////////////////////////////////
DRVLIST.PAS: Пример реализации нестандартной кнопки
/////////////////////////////////////////////////////////////////

```

```

uses Objects, App, Views, Drivers, Dialogs, Menus, MsgBox;
Const
  cmDialog = 999;
  cmSysMenu = 1000;
  cmDrvBase = 2000;

```

```

Var
  DColl : PStringCollection;

```

```

Type
  PDrvButton = ^TDrvButton;
  TDrvButton = Object(TView)
    Title : Char;
    constructor Init(var Bounds : TRect; ATitle : Char);
    procedure Draw; virtual;
    function GetPalette : PPalette; virtual;
    procedure HandleEvent(var Event : TEvent); virtual;
  End;

```

```

  PDrvDialog = ^TDrvDialog;
  TDrvDialog = Object(TDialog)
    procedure HandleEvent(var Event : TEvent); virtual;
  End;

```

```

  TMyApp = Object(TApplication)
    constructor Init;
    procedure InitStatusLine; virtual;
    procedure HandleEvent(var Event : TEvent); virtual;
    procedure NewDialog;
  End;

```

```

Function DriveValid(Drive: Char): Boolean; assembler;

```

```

asm
  MOV     AH,19H      Save the current drive in BL }
INT      21H
MOV      BL,AL
MOV      DL,Drive     { Select the given drive }
SUB      DL,'A'
MOV      AH,0EH
INT      21H
MOV      AH,19H       { Retrieve what DOS thinks is current }
INT      21H
MOV      CX,0         { Assume false }
CMP      AL,DL        { Is the current drive the given drive? }
JNE      @@@1
MOV      CX,1         { It is, so the drive is valid }
MOV      DL,BL        { Restore the old drive }
MOV      AH,0EH
INT      21H
@@@1:    XCHG         AX,CX      { Put the return value into AX }
end;

```

```

Constructor TDrvButton.Init;
Begin
  Bounds.B.X := Bounds.A.X + 3;
  Bounds.B.Y := Bounds.A.Y + 1;

```

```

TView.Init(Bounds);
Title := ATitle;
End;

Procedure TDrvButton.Draw;
Var
  B      : TDrawBuffer;
  I      : Byte;
  CButton : Word;
Begin
  CButton := GetColor($0501);
  MoveChar(B[0],',', CButton,1);
  MoveChar(B[1],Title,CButton,1);
  MoveChar(B[2],',', CButton,1);
  WriteLine(0,0,3,1,B);
End;

Function TDrvButton.GetPalette;
Const
  P : String[Length(CButton)] = CButton;
Begin
  GetPalette := @P;
End;

Procedure TDrvButton.HandleEvent;
Begin
  Inherited HandleEvent(Event);
  If Event.What = evMouseDown Then
    Begin
      If MouseInView(Event.Where) Then
        Begin
          Event.What := evCommand;
          Event.Command := cmDrvBase + Ord(Title)-Ord('A');
          Event.InfoPtr := Owner;
          PutEvent(Event);
          ClearEvent(Event);
        End;
      End Else ClearEvent(Event)
    End;
End;

Procedure TDrvDialog.HandleEvent;
Begin
  Inherited HandleEvent(Event);
  if Event.What = evCommand
  Then
    Case
      Event.Command of
        cmDrvBase..cmDrvBase + 26 :
          MessageBox(' C'You selected : ' + #13^C'Drive ' +
            Chr(Event.Command-cmDrvBase + Ord('A')),
            Nil,mfOkButton OR mfInformation)
        Else Exit;
      End;
    ClearEvent(Event);
  End;

Constructor TMyApp.Init;
Var
  Drive : Char;
Begin

```

```

Inherited Init;
{Коллекция символов дисков, доступных в системе}
DColl := New(PStringCollection, Init(26, 0));
For Drive := 'A' to 'S' do
  If DriveValid(Drive) Then DColl^.Insert(NewStr(Drive));
End;

```

```

Procedure TMyApp.InitStatusLine;
Var
  R : TRect;
Begin
  GetExtent(R);
  R.A.Y := R.B.Y - 1;
  StatusLine := New(PStatusLine, Init(R,
    NewStatusDef(0, $FFFF,
      NewStatusKey('~Alt-X~ Exit', kbAltX, cmQuit,
        NewStatusKey('~Alt-D~ Dialog', kbAltD, cmDialog,
          Nil)),
    Nil)
  ));
End;

```

```

Procedure TMyApp.HandleEvent;
Begin
  Inherited HandleEvent(Event);
  If Event.What = evCommand then
  Begin
    Case Event.Command of
      cmDialog : NewDialog;
    Else
      Exit;
    End;
    ClearEvent(Event);
  End;
End;

```

```

Procedure TMyApp.NewDialog;
Var
  Dialog : PDrvDialog;
  B      : PDrvButton;
  R      : TRect;
  I      : Byte;
  Drive  : PString;
Begin
  R.Assign(0, 1, (DColl^.Count * 3) + 2, 4);
  Dialog := New(PDrvDialog, Init(R, ""));
  Dialog^.Flags := 0;
  Dialog^.SetState(sfShadow, False);

```

```

With Dialog^ do
Begin
  R.A.X := 1;
  For I := 0 to DColl^.Count-1 do
  Begin
    R.A.Y := 1;
    Drive := DColl^.At(I);
    B := New(PDrvButton, Init(R, Drive^[1]));
    Insert(B);
    R.A.X := R.A.X + 3;
  End;
End;

```

```

End;
DeskTop^.ExecView(Dialog);
End;

Var
  MyApp : TMyApp;

Begin
  MyApp.Init;
  MyApp.Run;
  MyApp.Done;
End.

```

В приведенном выше примере показана реализация кнопки *TDrvButton*, которая отображает символ одного из дисков, доступных в системе, и при ее нажатии возвращает этот символ.

Объект TCluster

Кнопки с зависимой и независимой фиксацией

Помимо обычных кнопок (объект *TButton*), в Turbo Vision возможно использование кнопок с зависимой и независимой фиксацией. Для этого используются два объекта-наследника объекта *TCluster*: *TCheckBoxes* и *TRadioButton*s.

Объект *TCluster* является групповым объектом, который позволяет объединять несколько схожих объектов. Этот объект является абстрактным - его экземпляры не создаются. Он используется для задания свойств объектов-наследников: объектов *TCheckBoxes* и *TRadioButton*s. Обычно, объект типа *TCluster* ассоциируется с объектом *TLabel*, позволяющим выбрать необходимую группу объектов. Тогда как обычные кнопки используются для отправки определенных сообщений, кнопки на основе объекта *TCluster* используются для изменения значений поля *Value* (типа *Word*). Два стандартных наследника объекта *TCluster* изменяют значение поля *Value* по-разному: объект *TCheckBoxes* просто изменяет значение определенного бита, тогда как объект *TRadioButton*s изменяет значение определенного бита и переключает значение ранее выбранного.

Группа кнопок независимо от типа создается при помощи серии вложенных вызовов функции *NewSItem*:

```

R.Assign(10,5,40,20);
Control := New(PRadioButtons, Init(R,
NewSItem('~0~',
NewSItem('~1~',
NewSItem('~2~',
.....
NewSItem('~9~',nil)
....));

```

Необходимо также определить специальный тип данных для хранения значений данной группы кнопок:

```

TDialogData = Record
  RadioButtonsData : Word;
End;

```

После этого, подразумевая, что группа кнопок помещена в панель диалога, необходимо указать на область хранения данных:

```
Dialog^.SetData(DialogData)
```

Можно установить начальные значения группы кнопок, например:

```
DialogData.RadioButtonsData := 8;
```

После завершения работы панели диалога (ее закрытия), состояние кнопок определяется с помощью метода *GetData*:

```
Dialog^.GetData(DialogData)
```

Чтобы группа кнопок имела более привлекательный вид, их можно ограничить рамкой. Для этого необходимо установить соответствующий атрибут:

```
Control^.Options := Control^.Options OR ofFramed
```

После этого можно поместить название группы прямо на рамку. Для этого необходимо задать координаты объекта *TLabel* следующим образом:

```

R.Assign(Control^.Origin.X, Control^.Origin.Y-1,
Length(Text)-1, Control^.Origin.Y);

```

Пример создания группы кнопок с рамкой и заголовком приведен ниже:

```

////////////////////////////////////
BTNGROUP.PAS: Пример создания группы кнопок с рамкой и
заголовком
////////////////////////////////////
uses Dos, Objects, App, Dialogs, Views, Menus, Drivers, MsgBox;

```

```

Const
  cmNewDlg = 100;
Var
  R      : TRect;
Type
  TApp = Object(TApplication)
    Dialog : PDialog;
    procedure InitStatusLine;          virtual;
    procedure HandleEvent(var Event : TEvent);  virtual;
    procedure NewDlg;
  End;

```

```

Procedure TApp.InitStatusLine;
Begin
  GetExtent(R);
  R.A.Y := R.B.Y - 1;
  StatusLine := New(PStatusLine, Init(R,
    NewStatusDef(0, $FFFF,
      NewStatusKey('~Alt-X~ Exit', kbAltX, cmQuit,
        NewStatusKey('~Alt-D~ Dialog', kbAltD, cmNewDlg,
          Nil)),
    Nil)
  ));
End;
Procedure TApp.NewDlg;
Var
  Buttons : PRadioButtons;
  Text    : PLabel;
Begin
  R.Assign(0, 0, 40, 15);
  Dialog := New(PDialog, Init(R, 'Framed Group'));
  With Dialog do
    Begin
      Options := Options OR ofCentered;
      R.Assign(5, 5, 35, 8);
      Buttons := New(PRadioButtons, Init(R,
        NewSItem('Option 1',
        NewSItem('Option 2',
        NewSItem('Option 3',
        NewSItem('Option 4',
        NewSItem('Option 5',
        NewSItem('Option 6',
        nil))))))
      ));
      Buttons.Options := Buttons.Options OR ofFramed;
      Insert(Buttons);
      R.Assign(Buttons.Origin.X, Buttons.Origin.Y -
        1, 13, Buttons.Origin.Y);
      Text := New(PLabel, Init(R, 'Options', Buttons));
      Insert(Text);
    End;
  Application.InsertWindow(Dialog);
End;

```

```

Procedure TApp.HandleEvent;
Begin
  inherited HandleEvent(Event);
  if Event.What = evCommand Then
    Begin
      Case Event.Command of
        cmNewDlg : NewDlg;
        else Exit;
      End;
    End;
  ClearEvent(Event);
End;
Var DApp : TApp;
Begin
  DApp.Init;
  DApp.Run;
  DApp.Done;
End.

```

При необходимости можно создать объект-наследник объекта *TCheckBoxes* или *TRadioButtons*, конструктор которого будет выполнять все необходимые действия. Для этого нужно ввести еще один параметр конструктора: название заголовка. Реализацию этого объекта оставляем читателю в качестве упражнения.

Еще одной практичной возможностью является возможность запрета использования отдельных кнопок в группе. В Turbo Vision версии 2.0 объект *TCluster* (который является предком объектов *TCheckBoxes* и *TRadioButtons*) имеет поле *EnableMask* типа *LongInt*. По умолчанию используется значение *\$FFFFFFFF*. Каждый байт поля *EnableMask* отвечает за 32 элемента группы. Таким образом, можно разрешить/запретить использование любого элемента в группе кнопок. Например, если значение этого поля равно 7, то доступны только первые три элемента. Если значение поля *EnableMask* равно 0, то не может быть выбрана вся группа кнопок.

Изменения в Turbo Vision 2.0

Объект *TCluster*

Объект *TCluster* содержит ряд методов, позволяющих реализовать элементы управления со многими состояниями.

Поле *EnableMask*

Значение этого поля позволяет задать состояние первых 32 элементов управления в группе. Если бит установлен, элемент управления активен и может быть использован.

Метод ButtonState

С помощью этого метода можно определить текущее состояние элемента управления в группе.

Метод DrawMultiBox

С помощью этого метода осуществляется отображение кнопки со многими состояниями.

Метод MultiMark

С помощью этого метода можно определить текущее состояние элемента со многими состояниями в группе подобных элементов.

Метод SetButtonState

Этот метод позволяет изменять значения поля *EnableMask*.

Объект TListBox

Для создания списков в Turbo Vision используется объект *TListBox*. При инициализации списка указывается размер, число колонок (список может содержать более одной колонки) и необязательная полоса прокрутки. Конструктор *Init* выполняет следующие действия: вызывает конструктор объекта-предка, присваивает указателю на коллекцию значение *Nil* и устанавливает размер списка, равным нулю. Содержимое списка задается с помощью метода *NewList*, в котором в качестве параметра указывается указатель на коллекцию, содержащую список строк, которые должны быть отображены. Можно использовать как обычную коллекцию (объект *TCollection*), так и отсортированную коллекцию (объект *TSortedCollection*). Удобно создать специальную функцию, которая будет возвращать указатель на коллекцию и использовать вызов этой функции в качестве параметра метода *NewList*:

```
ListBox .NewList(BuildList)
```

Метод *NewList* присваивает значение указателя на коллекцию полю *List*, устанавливает диапазон элементов списка :

```
SetRange(AList.Count)
```

и устанавливает фокус на первый элемент списка:

```
FocusItem(0)
```

Совместно со списками можно использовать специальную запись типа *TListBoxRec*, которая состоит из двух полей: поля *List* типа *PCollection* и поля *Selection* типа *Word*. Запись этого типа может использоваться совместно с методами *GetData* и *SetData* для установки начальных значений и получения результатов выбора.

Для получения результата выбора, необходимо переопределить метод *HandleEvent*. В приведенном ниже примере показано, как в созданном на базе объекта *TListBox* объекте *TMLListBox* переопределен метод *HandleEvent*. В нем обрабатываются два события: нажатие клавиши Enter и двойное нажатие кнопки мыши. В ответ на эти события отображается панель сообщений, в которой указывается текущий выбранный элемент. Также, можно использовать переопределение метода *HandleEvent* для отправки какого-либо сообщения объекту-владельцу списка:

```
Event.What = evCommand;  
Event.Command = cmListBox;  
PutEvent(Event);
```

Метод *HandleEvent* владельца должен обрабатывать это сообщение. Также, возможно использование функции *Message*.

В приведенном ниже примере показано, как создать панель диалога, содержащую список с элементами, отсортированными в алфавитном порядке. Выбор элемента списка осуществляется при нажатии клавиши Enter или двойном нажатии кнопки мыши.


```

Selection : Word;
End;

```

```

Var
ListBoxRec : TListBoxRec;
LB          : PMLListBox;

```

```

Function TSCollection.Compare;
Begin
If PString(Key1) < PString(Key2) Then Compare := -1
else
If PString(Key1) > PString(Key2) Then Compare := 1
else
Compare := 0;
End;

```

```

Procedure TMLListBox.HandleEvent;
Begin
If ((Event.What = evKeyDown) AND (Event.KeyCode =
kbEnter))
OR ((Event.What = evMouseDown) AND (Event.Double))
Then
Begin
MessageBox('C'You selected : ' + GetText(Focused,128),Nil,
mfOkButton);
End
Else Inherited HandleEvent(Event);
End;

```

```

Procedure TDemoApp.InitStatusLine;
Var
R : TRect;
Begin
GetExtent(R);
R.A.Y := R.B.Y - 1;
StatusLine := New(PStatusLine, Init(R,
NewStatusDef(0, $FFFF,
NewStatusKey('~Alt-X~ Exit', kbAltX, cmQuit,
NewStatusKey('~Alt-D~ Dialog', kbAltD, cmDialog,
Nil)),
Nil));
));
End;
Function TDemoApp.BuildList;
Var
List : PSCollection;
Begin
List := New(PSCollection, Init(10,10));
With List do
Begin
Insert(NewStr('Turbo Pascal'));
Insert(NewStr('Borland C + +'));
Insert(NewStr('Object Vision'));
Insert(NewStr('Paradox'));
Insert(NewStr('Quattro Pro'));
Insert(NewStr('Paradox Engine'));

```

```

Insert(NewStr('Paradox for Window'));
Insert(NewStr('SideKick'));
Insert(NewStr('InterBase'));
Insert(NewStr('Quattro Pro for Windows'));
Insert(NewStr('dBase 2.5'));
End;
BuildList := List;
End;

```

```

Procedure TDemoApp.HandleEvent;
Begin
  Inherited HandleEvent(Event);
  if Event.What = evCommand then
  begin
    case Event.Command of
      cmDialog : DemoDialog;
    else
      Exit;
    end;
    ClearEvent(Event);
  end;
End;
Procedure TDemoApp.DemoDialog;
Var
  Dialog : PDialogBox;
  R      : TRect;
  SB     : PScrollBar;
Begin
  R.Assign(20,5,60,20);
  Dialog := New(PDialogBox, Init(R, 'Demo Dialog'));
  With Dialog do
  Begin
    R.Assign(10,2,30,3);
    Insert(New(PStaticText, Init(R, 'C' Borland Products')));
    {Полоса прокрутки}
    R.Assign(36,3,37,13);
    SB := New(PScrollBar, Init(R));
    Insert(SB);
    {Список}
    R.Assign(5,3,35,13);
    LB := New(PListBox, Init(R, 1, SB));
    ListBoxRec.List := BuildList;
    ListBoxRec.Selection := 0;
    LB.SetData(ListBoxRec);
    LB.Options := LB.Options OR ofFramed;
    Insert(LB);
    Palette := dpCyanDialog;
  End;
  ExecuteDialog(Dialog, Nil);
End;

```

```

Procedure TDialogBox.HandleEvent;
Begin
  Inherited HandleEvent(Event);
End;

```

```

Var
  DemoApp : TDemoApp;
Begin
  DemoApp.Init;
  DemoApp.Run;
  DemoApp.Done;
End.

```

Отметим, что в методе *TMListBox.HandleEvent* текущий выбор осуществляется с помощью метода *GetText*, возвращающего содержимое элемента, который находится в фокусе (содержимое поля *Focused*). Для создания списка элементов используется функция *TDemoApp.BuildList*, возвращающая указатель на отсортированную коллекцию (тип *TSCollection*). Для установки начальных значений списка используется специальная запись (тип *TListBoxRec*) и метод *TMListBox.SetData*. Отметим создание полосы прокрутки для списка. Распространенной ошибкой является то, что пользователи забывают вызвать метод *Insert* после создания полосы прокрутки, полагая, что указания на нее при создании списка вполне достаточно.

Список с возможностью выбора нескольких элементов

Объект *TListBox* обладает существенным недостатком: в нем можно выбрать только один элемент. Ниже проиллюстрировано, как расширить свойства объекта *TListBox* таким образом, чтобы пользователь мог одновременно выбрать (пометить) несколько элементов. В качестве клавиши для пометки элементов выберем клавишу *Ins*, а в качестве индикатора - символ '№'. Очевидно, что нам необходимо переопределить метод *HandleEvent*, чтобы мы могли обрабатывать нажатие клавиши *Ins* и метод *Draw* для отображения помеченных элементов. Для простоты реализуем пометку элементов списка следующим образом: к каждому помеченному элементу будем добавлять в первую позицию текста символ '№': это облегчит задачу хранения помеченных элементов. Также, заведем клавишу (в данной реализации - *Enter*), нажатие которой будет возвращать владельцу объекта указатель на список, из которого необходимо будет извлечь помеченные элементы (т.е. те элементы, у которых первый символ равен символу '№').

Посмотрим на объект *TListBox* и его предок - объект *TListViewer*. Поле *Focused* позволяет нам определить элемент списка, который находится в фокусе, поле *Range* - число элементов в списке. Введем еще одно допущение: при нажатии клавиши *Ins* фокус будет перемещаться на следующий элемент, при нажатии этой клавиши на последнем элементе списка фокус будет перемещаться на первый элемент. Процедура пометки элемента списка будет выглядеть следующим образом:

```

Procedure CheckItem;
Begin
  Text := GetText(Focused,Size.X);
  {Элемент уже выбран ?}
  If Text[1] = MarkChar Then
    Begin
      Text := GetItemText(Focused,Size.X);
      List^.AtPut(Focused,NewStr(Text));
    End
  {Еще нет}
  Else
    List^.AtPut(Focused,NewStr(MarkChar + Text));
  {Переместить фокус на следующий элемент}
  If Focused + 1 < Range Then FocusItem(Focused + 1)
  Else FocusItem(0);
  DrawView;
End;

```

Сначала мы проверяем, имеется ли пометка в тексте элемента и если она есть - удаляем ее, в противном случае мы помещаем метку и заменяем соответствующий элемент коллекции. После этого вызывается метод *DrawView*, который выполняет перерисовку всего объекта.

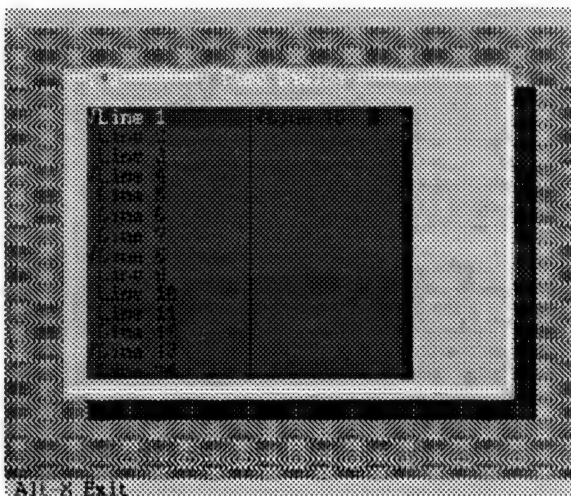


Рис.3.4.Реализация объекта TMSListBox приведена ниже.

```

////////////////////////////////////
TMSListBox: Список с возможностью выбора нескольких
элементов
//////////////////////////////////// }
    unit MSLBOX;
    Interface
    uses Dialogs, Drivers;
    Const
        cmListSelected = 3000; {Пользователь сделал выбор}
    Var
        MarkChar : Char;      {Символ пометки}
    Type
        PMSListBox = ^TMSListBox;
        TMSListBox = Object(TListBox)
        function GetItemText(Item : Integer; MaxLen : Integer) :
            String;
        procedure HandleEvent(Var Event : TEvent); virtual;
        procedure Draw;      virtual;
    End;

    Implementation
    uses Objects, Views;
    {
        TMSListBox Object
    }
    Function TMSListBox.GetItemText;
    Var
        Text : String;
    Begin
        Text := GetText(Item, MaxLen);
        {Удалить пометку}
        If Text[1] = MarkChar Then System.Delete(Text, 1, 1);

```

```

GetItemText := Text;
End;
Procedure TMSListBox.HandleEvent;
Var
  Text : String;

Procedure CheckItem;
Begin
  Text := GetText(Focused,Size.X);
  {Элемент уже выбран ?}
  If Text[1] = MarkChar Then
    Begin
      Text := GetItemText(Focused,Size.X);
      List^.AtPut(Focused,NewStr(Text));
    End
  Else
    List^.AtPut(Focused,NewStr(MarkChar + Text));
  If Focused + 1 < Range Then FocusItem(Focused + 1)
  Else FocusItem(0);
  DrawView;
End;
{!! .01}
Procedure CheckWithMouse;
Var
  Mouse : TPoint;
  ColWidth : Word;
  OldItem,
  NewItem : Integer;
  Count : Word;
  Text : String;
Begin
  ColWidth := Size.X DIV NumCols + 1;
  OldItem := Focused;
  MakeLocal(Event.Where, Mouse);
  NewItem := Mouse.Y + (Size.Y * (Mouse.X DIV ColWidth))
    + TopItem;
  If NewItem <= Range-1 Then
    Begin;
      Text := GetText(NewItem,Size.X);
      If Text[1] = MarkChar Then
        Begin
          Text := GetItemText(NewItem,Size.X);
          List^.AtPut(NewItem,NewStr(Text));
        End
      Else
        List^.AtPut(NewItem,NewStr(MarkChar + Text));
        FocusItem(NewItem);
        DrawView;
    End; {If NewItem <= Range-1}
End;
{!! .01}
{Handle Event}
Begin
  {Клавиша Ins ?}
  If ((Event.What = evKeyDown) AND (Event.KeyCode =
    kbIns)) Then
    Begin
      CheckItem;
      ClearEvent(Event);
    End;
  End;
End;

```

```

End;
{Клавиша Enter ?}
If ((Event.What = evKeyDown) AND (Event.KeyCode =
kbEnter)) Then
Begin
    Message(Owner, evBroadcast, cmListSelected, @Self);
    ClearEvent(Event);
End;
{!! 01}
{Правая кнопка мыши ?}
If ((Event.What = evMouseDown) AND (Event.Buttons =
mbRightButton)) Then
Begin
    CheckWithMouse;
    ClearEvent(Event);
End;
{!! .01}
Inherited HandleEvent(Event);
End;

procedure TMSListBox.Draw;
var
    I, J, Item: Integer;
    NormalColor, SelectedColor, FocusedColor, Color: Word;
    ColWidth, CurCol, Indent: Integer;
    B: TDrawBuffer;
    Text: String;
    SOff: Byte;
begin
    if State and (sfSelected + sfActive) = (sfSelected + sfActive)
    then
        begin
            NormalColor := GetColor(1);
            FocusedColor := GetColor(3);
            SelectedColor := GetColor(4);
        end else
        begin
            NormalColor := GetColor(2);
            SelectedColor := GetColor(4);
        end;
    if HScrollBar <> nil then Indent := HScrollBar.Value
    else Indent := 0;
    ColWidth := Size.X div NumCols + 1;
    for I := 0 to Size.Y - 1 do
        begin
            for J := 0 to NumCols-1 do
                begin
                    Item := J*Size.Y + I + TopItem;
                    CurCol := J*ColWidth;
                    if (State and (sfSelected + sfActive) = (sfSelected +
sfActive)) and
                        (Focused = Item) and (Range > 0) then
                        begin
                            Color := FocusedColor;
                            SetCursor(CurCol+1,I);
                            SOff := 0;
                        end
                    else if (Item < Range) and IsSelected(Item) then
                        begin

```

```

        Color := SelectedColor;
        SCOff := 2;
    end
    else
    begin
        Color := NormalColor;
        SCOff := 4;
    end;
    MoveChar(B[CurCol], ' ', Color, ColWidth);
    if Item < Range then
    begin
        Text := GetText(Item, ColWidth + Indent);
        Text := Copy(Text, Indent, ColWidth);
    (*)

    If Text[1] = MarkChar Then MoveStr(B[CurCol], Text, Color) Else
    (*)
        MoveStr(B[CurCol + 1], Text, Color);
        if ShowMarkers then
        begin
            WordRec(B[CurCol]).Lo := Byte(SpecialChars[SCOff]);
            WordRec(B[CurCol + ColWidth - 2]).Lo := Byte(SpecialChars[SCOff + 1]);
        end;
    end;
    MoveChar(B[CurCol + ColWidth - 1], #179, GetColor(5), 1);
    end;
    WriteLine(0, I, Size.X, 1, B);
end;
end;
Begin
    MarkChar := '№';
End.

```

Примечание: для реализации объекта *TMSListBox* пришлось полностью скопировать метод *TListBox.Draw* и добавить в него всего лишь одну строку (она помечена комментариями). Изменения заключаются в том, что если элемент списка имеет пометку, то он отображается не с первой позиции, а с нулевой.

Объект TStaticText

Для отображения текста в Turbo Vision используется объект *TStaticText*. Обычно, этот объект используется совместно с панелями диалога и окнами. Обычно, используется только конструктор *Init* этого объекта. Если параметр *Bounds* описывает область, содержащую более одной строки, текст может располагаться на нескольких строках. Если текст начинается с символа *^C* (*Chr(03)*), то он будет сцентрирован. Текст может быть разделен на несколько строк, если использовать символ *^M* (*Chr(13)*). Пример использования объекта *TStaticText* приведен ниже.

```

/////////////////////////////////////////////////////////////////
STATIC.PAS: Пример использования объекта TStaticText
/////////////////////////////////////////////////////////////////
uses Objects, Drivers, Views, Menus, Dialogs, App, MsgBox;

Const
  cmNewDialog = 101;

Type
  TMyApp = object(TApplication)
    procedure HandleEvent(var Event: TEvent); virtual;
    procedure InitStatusLine; virtual;
    procedure NewDialog;
  End;

Procedure TMyApp.HandleEvent(var Event: TEvent);
Begin
  TApplication.HandleEvent(Event);
  if Event.What = evCommand then
    begin
      case Event.Command of
        cmNewDialog: NewDialog;
      else Exit;
      end;
      ClearEvent(Event);
    end;
End;

Procedure TMyApp.InitStatusLine;
var R: TRect;
Begin
  GetExtent(R);
  R.A.Y := R.B.Y - 1;
  StatusLine := New(PStatusLine, Init(R,
    NewStatusDef(0, $FFFF,
      NewStatusKey('~Alt-X~ Exit', kbAltX, cmQuit,
        NewStatusKey('~Alt-D~ Dialog', kbAltD, cmNewDialog,
          nil)),
      nil)));
End;

Procedure TMyApp.NewDialog;
Var
  Dlg : PDialog;
  R : TRect;
  ST : PStaticText;
Begin
  R.Assign(20, 2, 60, 16);
  Dlg := New(PDialog, Init(R, 'Demo Dialog'));
  with Dlg do
    begin
      R.Assign(2, 3, 38, 4);
      {Обычный текст}
      ST := New(PStaticText, Init(R, 'Simple Text'));
      ST.Options := ST.Options OR ofFramed;
      Insert(ST);
      R.Assign(2, 5, 38, 6);
      {Сцентрированный текст}
    end
  end;
End;

```

```

ST := New(PStaticText, Init(R, ^C'Centered Text'));
ST^.Options := ST^.Options OR ofFramed;
Insert(ST);
R.Assign(2,7,38,10);
{Многострочный текст}
ST := New(PStaticText,
      Init(R, ^C'Multiline' + ^M^C'Centered' + ^M^C'Text'));
ST^.Options := ST^.Options OR ofFramed;
Insert(ST);
end;
DeskTop^.ExecView(Dlg);
End;

Var
  MyApp: TMyApp;

Begin
  MyApp.Init;
  MyApp.Run;
  MyApp.Done;
End.

```

Если вы проанализируете этот пример, то увидите, что каждый объект типа *TStaticText* имеет рамку. Это сделано для того, чтобы показать область экрана, занимаемую каждым объектом. Для того чтобы убрать рамку, необходимо удалить три строки:

```
ST^.Options := ST^.Options OR ofFramed;
```

Интерфейсные элементы, создаваемые на базе объекта *TStaticText*, являются примитивными отображаемыми объектами - они не реагируют на события и сами не являются источниками событий. Используя поле *Text* (типа *PString*), можно изменять содержимое этого объекта. После того как новое значение поля *Text* установлено, необходимо вызвать метод *Draw*:

```
ST^.Text := NewStr('New Text');
ST^.Draw;
```

Объект-наследник объекта *TStaticText* - *TLabel* обладает более интересными свойствами, которые рассматриваются ниже.

Объект TLabel

Объект *TLabel* обычно используется для задания заголовков других отображаемых объектов, например, совместно со строками ввода или группами кнопок. Объект

TLabel отличается от *TStaticText* только тем, что последний непосредственно связывается с элементом управления и используется для выбора этого элемента с помощью "мыши" или командной клавиши. При этом активизируется не заголовок, а связанный с ним элемент управления. Пример использования объекта *TLabel* рассмотрен ниже.

```
{//////////////////////////////////////////
LABEL.PAS: Пример использования объекта TLabel
//////////////////////////////////////////}
```

```
Const
  cmNewDialog = 101;
```

```
Type
  TMyApp = object(TApplication)
    procedure HandleEvent(var Event: TEvent); virtual;
    procedure InitStatusLine; virtual;
    procedure NewDialog;
  End;
```

```
Procedure TMyApp.HandleEvent(var Event: TEvent);
Begin
  TApplication.HandleEvent(Event);
  if Event.What = evCommand then
    begin
      case Event.Command of
        cmNewDialog: NewDialog;
      else Exit;
      end;
      ClearEvent(Event);
    end;
End;
```

```
Procedure TMyApp.InitStatusLine;
var R: TRect;
Begin
  GetExtent(R);
  R.A.Y := R.B.Y - 1;
  StatusLine := New(PStatusLine, Init(R,
    NewStatusDef(0,$FFFF,
      NewStatusKey('~Alt-X~ Exit',kbAltX,cmQuit,
        NewStatusKey('~Alt-D~ Dialog',kbAltD,cmNewDialog,
          nil)),
      nil)));
End;
```

```
Procedure TMyApp.NewDialog;
Var
  Dlg : PDialog;
  R : TRect;
  IL : PInputLine;
Begin
  R.Assign(20,2,60,16);
  Dlg := New(PDialog,Init(R,'Demo Dialog'));
  with Dlg do
    begin
```

```

R.Assign(5,5,35,6);
IL := New(PInputLine, Init(R,60));
Insert(IL);
R.Assign(5,4,35,5);
Insert(New(PLabel, Init(R,'I~nput Line', IL)));
R.Assign(5,7,35,8);
IL := New(PInputLine, Init(R,60));
Insert(IL);
R.Assign(5,6,35,7);
Insert(New(PLabel, Init(R,'Input ~L~ine', IL)));
end;
DeskTop^.ExecView(Dlg);
End;

Var
MyApp: TMyApp;

Begin
MyApp.Init;
MyApp.Run;
MyApp.Done;
End.

```

В приведенном примере проиллюстрировано создание двух строк ввода и двух объектов *TLabel*, связанных с каждой строкой. Первая строка может быть выбрана командной клавишей Alt-I, а вторая - командной клавишей Alt-L. Если параметр *ALink* при инициализации объекта *TLabel* равен *Nil*, объект *TLabel* используется как объект *TStaticText*.

Объект TParamText

Для отображения обычного текста используется объект *TStaticText*, для отображения заголовков элементов управления - объект *TLabel*, а для отображения параметризованного текста может использоваться объект *TParamText*. Пример использования этого объекта приведен ниже.

```

Var
AParam : PParamText;
Parameters : record
  AParamStr : PString;
End;
...
{ Установить значения панели диалога Delete }
with Dialog^ do
Begin
  ...
  Bounds.Assign (8,2,45,3);
  AParam := New (PParamText,
    Init( Bounds, 'Okay to Delete %s', 1 ));

```

```

Parameters.AParamStr := NewStr(FileName);
AParam^.ParamList := @Parameters;
Insert( AParam );
End;
Control := DeskTop^.ExecView (Dialog);

```

Объект *THistory*

Объект *THistory* используется для ведения протокола: сохранения списка ранее введенной в строке ввода информации. Обычно такой объект отображается в виде символа "стрелка вниз" справа от строки ввода, с которой он связан. Каждый раз при вводе новой информации в строке, предыдущее значение сохраняется в протоколе. Когда требуется просмотр или использование содержимого протокола, необходимо нажать "стрелку вниз" или активизировать символ протокола мышью - в ответ на эти действия появится небольшой список, отображающий содержимое протокола. Обычно все, что требуется для использования объекта *THistory* - это вызвать его конструктор и поместить сам объект в панель диалога. Действия по ведению протокола выполняются автоматически. Пример использования объекта *THistory* совместно со строкой ввода (объект *TInputLine*) рассмотрен ниже.

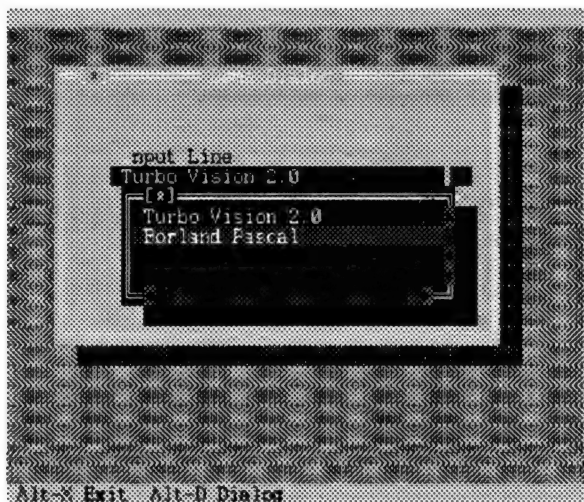


Рис.3.5.Использование протокола

```

{////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////}
HISTORY.PAS: Пример использования объекта THistory
{////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////}

```

```

uses Objects, Drivers, Views, Menus, Dialogs, App, MsgBox;

```

```

Const
  cmNewDialog = 101;

```

```

Type
  TMyApp = Object(TApplication)
    procedure HandleEvent(var Event: TEvent); virtual;
    procedure InitStatusLine; virtual;
    procedure NewDialog;
  End;

```

```

Procedure TMyApp.HandleEvent(var Event: TEvent);
Begin
  TApplication.HandleEvent(Event);
  if Event.What = evCommand then
    begin
      case Event.Command of
        cmNewDialog: NewDialog;
        else Exit;
      end;
      ClearEvent(Event);
    end;
End;

```

```

Procedure TMyApp.InitStatusLine;
var R: TRect;
Begin
  GetExtent(R);
  R.A.Y := R.B.Y - 1;
  StatusLine := New(PStatusLine, Init(R,
    NewStatusDef(0, $FFFF,
      NewStatusKey('~Alt-X~ Exit', kbAltX, cmQuit,
        NewStatusKey('~Alt-D~ Dialog', kbAltD, cmNewDialog,
          nil)),
      nil)));
End;

```

```

Procedure TMyApp.NewDialog;
Var
  Dlg : PDialog;
  R : TRect;
  IL : PInputLine;
  H : PHistory;
Begin
  R.Assign(20, 2, 60, 16);
  Dlg := New(PDialog, Init(R, 'Demo Dialog'));
  with Dlg do
    begin
      R.Assign(5, 5, 35, 6);
      IL := New(PInputLine, Init(R, 60));
      Insert(IL);
      R.Assign(5, 4, 35, 5);
      Insert(New(PLabel, Init(R, 'Input Line', IL)));
      R.Assign(35, 5, 38, 6);
      Insert(New(PHistory, Init(R, IL, 1)));
    end;
  end;

```

```

end;
DeskTop^.ExecView(Dlg);
End;

Var
  MyApp: TMyApp;

Begin
  MyApp.Init;
  MyApp.Run;
  MyApp.Done;
End.

```

Несмотря на то что такое использование протокола может удовлетворить большинство разработчиков, мы рассмотрим некоторые способы улучшения внешнего вида протокола. Как вы можете заметить, при выполнении данной программы, протокол при своем отображении закрывает строку ввода. Это происходит из-за того, что при инициализации окна, в котором отображается протокол, координаты задаются, на мой взгляд, не совсем правильным образом. Чтобы расположить окно протокола более удобным образом, скажем, на одну строку ниже связанного с ним интерфейсного элемента и на один символ правее, необходимо переопределить (а точнее, переписать) метод *THistory.HandleEvent* и заменить в нем строку:

```
Dec(R.A.X); Inc(R.B.X); Inc(R.B.Y, 7); Dec(R.A.Y, 1)
```

на

```
Inc(R.A.X); Inc(R.B.X); Inc(R.B.Y, 7); Inc(R.A.Y, 1);
```

Также можно изменить символ, с помощью которого вызывается окно просмотра протокола. Для этого, у объект *THistory* необходимо изменить метод *HandleEvent*, метод *Draw*, а также убедиться в том, что выбранный символ игнорируется строкой ввода (как в случае со "стрелкой вниз"). Пример переопределенного метода *Draw* приведен ниже.

```

Procedure TNewHistory.Draw;
Var
  B : TDrawBuffer;
Begin
  MoveCStr(B, #222"~""#221, GetColor($0102));
  WriteLine(0, 0, Size.X, Size.Y, B);
End;

```

Если мы хотим, чтобы окно просмотра протокола вызывалось по нажатию символа "~", необходимо создать объект-наследник объекта *TInputLine* и заставить его

игнорировать ввод символа "*" Как это сделать, проиллюстрировано ниже.

```

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
HISTORY1.PAS: Пример использования объекта THistory.
Приведен пример переопределения методов
THistory.HandleEvent, THistory.Draw и создания
специализированной строки ввода - объект TNewInoutLine
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////}
uses Objects, Drivers, Views, Menus, Dialogs, App, MsgBox;

Const
  cmNewDialog = 101;

Type
  TMyApp = Object(TApplication)
    procedure HandleEvent(var Event: TEvent); virtual;
    procedure InitStatusLine; virtual;
    procedure NewDialog;
  End;

  PNewHistory = ^TNewHistory;
  TNewHistory = Object(THistory)
    procedure HandleEvent(var Event: TEvent); virtual;
    procedure Draw; virtual;
  End;

  PNewInputLine = ^TNewInputLine;
  TNewInputLine = Object(TInputLine)
    procedure HandleEvent(var Event: TEvent); virtual;
  End;

Procedure TNewInputLine.HandleEvent;
Begin
  TView.HandleEvent(Event);
  If ((Event.What = evKeyDown) AND (Event.CharCode = '*'))
  Then
    Else Inherited HandleEvent(Event);
End;

Procedure TNewHistory.HandleEvent;
Var
  HistoryWindow: PHistoryWindow;
  R, P: TRect;
  C: Word;
  Rslt: String;
Begin
  TView.HandleEvent(Event);
  If (Event.What = evMouseDown) OR
    ((Event.What = evKeyDown) AND (Event.CharCode = '*')
    AND
    (Link^.State AND sfFocused <> 0)) then
  begin
    if not Link^.Focus then
    begin
      ClearEvent(Event);
      Exit;
    end;
  end;

```

```

RecordHistory(Link^.Data^);
Link^.GetBounds(R);
Inc(R.A.X); Inc(R.B.X); Inc(R.B.Y,7); Inc(R.A.Y,1);
Owner^.GetExtent(P);
R.Intersect(P);
Dec(R.B.Y,1);
HistoryWindow := InitHistoryWindow(R);
if HistoryWindow <> Nil then
begin
  C := Owner^.ExecView(HistoryWindow);
  if C = cmOk then
  begin
    Rslt := HistoryWindow^.GetSelection;
    If Length(Rslt) > Link^.MaxLen then Rslt[0] :=
      Char(Link^.MaxLen);
    Link^.Data^ := Rslt;
    Link^.SelectAll(True);
    Link^.DrawView;
  end;
  Dispose(HistoryWindow, Done);
end;
ClearEvent(Event);
end
else if (Event.What = evBroadcast) then
if ((Event.Command = cmReleasedFocus) AND
(Event.InfoPtr = Link))
OR (Event.Command = cmRecordHistory) then
RecordHistory(Link^.Data^);
End;
Procedure TNewHistory.Draw;
Var
  B : TDrawBuffer;
Begin
  MoveCStr(B, #222'~'~'~' #221, GetColor($0102));
  WriteLine(0, 0, Size.X, Size.Y, B);
End;

Procedure TMyApp.HandleEvent(var Event: TEvent);
Begin
  TApplication.HandleEvent(Event);
  if Event.What = evCommand then
  begin
    case Event.Command of
      cmNewDialog: NewDialog;
    else Exit;
    end;
    ClearEvent(Event);
  end;
End;

Procedure TMyApp.InitStatusLine;
var R: TRect;
Begin
  GetExtent(R);
  R.A.Y := R.B.Y - 1;
  StatusLine := New(PStatusLine, Init(R,
    NewStatusDef(0,$FFFF,
    NewStatusKey('~Alt-X~ Exit',kbAltX,cmQuit,
    NewStatusKey('~Alt-D~ Dialog',kbAltD,cmNewDialog,
    nil))),

```

```

    nil));
End;

Procedure TMyApp.NewDialog;
Var
  Dlg    : PDialog;
  R      : TRect;
  IL     : PNewInputLine;
  H      : PNewHistory;
Begin
  R.Assign(20,2,60,16);
  Dlg := New(PDialog, Init(R, ""));
  with Dlg do
  begin
    R.Assign(5,5,35,6);
    IL := New(PNewInputLine, Init(R,60));
    Insert(IL);
    R.Assign(5,4,35,5);
    Insert(New(PLabel, Init(R, "Input Line", IL)));
    R.Assign(35,5,38,6);
    Insert(New(PNewHistory, Init(R, IL, 1)));
  end;
  DeskTop^.ExecView(Dlg);
End;

Var
  MyApp: TMyApp;

Begin
  MyApp.Init;
  MyApp.Run;
  MyApp.Done;
End.

```

Объект *TNewInputLine* реализует строку ввода, которая игнорирует ввод символа "*". Объект *TNewHistory* - это наследник объекта *THistory*, который изменяет следующие свойства своего предка: отображение окна протокола происходит при нажатии клавиши "*", и окно протокола отображается на строку ниже строки ввода и сдвинутое на одну позицию вправо. Это достигается переопределением методов *HandleEvent* и *Draw*.

На мой взгляд, реализация метода *THistory.HandleEvent* не совсем удачна: для создания окна просмотра протокола используется метод *InitHistoryWindow*. Более логичным было бы внести в этот метод вычисление координат окна, а не выполнять эти вычисления в методе *HandleEvent*. Тогда для того чтобы отобразить окно в другом месте, нам было бы необходимо только переопределить метод *InitHistoryWindow*, не трогая метод *HandleEvent*. Метод *InitHistoryWindow* должен выглядеть следующим образом:

```

Function TNewHistory.InitHistoryWindow;
Var

```

```

P,R : TRect;
H : PHistoryWindow;
Begin
{**}
  Link^.GetBounds(R);
  Inc(R.A.X); Inc(R.B.X); Inc(R.B.Y,7); Inc(R.A.Y,1);
  Owner^.GetExtent(P);
  R.Intersect(P);
  Dec(R.B.Y,1);
{**}
  H := New(PHistoryWindow, Init(R, HistoryID));
  H.HelpCtx := Link.HelpCtx;
  InitHistoryWindow := H;
End;

```

Строки, заключенные между комментариями **{**}**, перенесены из метода *HandleEvent* и должны быть удалены в этом методе. Таким образом, в следующий раз, когда вам потребуется изменить местоположение окна просмотра протокола, необходимо будет только переопределить метод *InitHistoryWindow* у объекта *TNewHistory*.

Использование протокола

Объект *TInputLine* используется совместно со строками ввода для реализации протокола вводимых данных. Этот протокол позволяет хранить ранее введенные строки и повторно их использовать. Не совсем очевидным свойством протокола является то, что можно изначально задать его содержимое, облегчив тем самым ввод часто используемых строк. Для установки начальных значений применяется глобальная процедура *HistoryAdd*. Все, что требуется выполнить, - это добавить к протоколу необходимые строки. Ниже приводится фрагмент программы, реализующей сказанное:

```

R.Assign(9,2,38,3);
Line := New(PInputLine, Init(R,29));
Insert(Line);
R.Assign(39,2,41,3);
Control := New(PNewHistory, Init(R,Line,100));
Insert(Control);
S := 'Turbo Pascal 5.5'; HistoryAdd(100,S);
S := 'Turbo Pascal 6.0'; HistoryAdd(100,S);
S := 'Turbo Pascal for Windows 1.5'; HistoryAdd(100,S);
.....
S := 'Object Vision 2.1 Pro'; HistoryAdd(100,S);

```

Заметим, что размер окна, в котором отображается протокол, зависит от положения связанной с ним строки ввода внутри панели диалога. Для изменения внешнего вида

кнопки вызова протокола необходимо переопределить метод *THistory.Draw*, а для изменения командной клавиши и положения протокола относительно строки ввода метод *THistoryViewer.HandleEvent*.

Изменения в Turbo Vision 2.0 *Объект THistory*

Метод RecordHistory

Позволяет добавить строку к протоколу. Используется текущий протокол (поле *HistoryId*). Действие этого метода эквивалентно вызову глобальной процедуры:

```
HistoryAdd(HistoryId, S)
```

Объект TInputLine

Объект *TInputLine* используется для создания строк ввода с возможностью редактирования. Объект *TinputLine* реализует однострочный редактор с возможностью горизонтального скроллинга, выделения текста и поддержания блочных операций. Если размер строки ввода меньше длины текста, выполняется автоматический скроллинг вправо и влево.

В большинстве случаев используется только конструктор *TinputLine.Init*, с помощью которого и создается строка ввода. Строка ввода становится активной при получении фокуса и весь ввод обрабатывается методом *HandleEvent*. Тогда как по умолчанию поддерживается ввод символьной информации, переопределение ряда методов позволит вводить другие типы данных. Ниже приведен пример использования строки ввода совместно с объектом *TLabel*, который служит для указания на тип вводимой информации и как средство для выбора строки ввода.

```
{//////////////////////////////////////  
INPUT.PAS: Пример использования объекта TinputLine  
//////////////////////////////////////}  
uses Objects, Drivers, Views, Menus, Dialogs, App, MsgBox;  
  
Const  
  cmNewDialog = 101;  
  
Type
```

```

TMyApp = Object(TApplication)
  procedure HandleEvent(var Event: TEvent); virtual;
  procedure InitStatusLine; virtual;
  procedure NewDialog;
End;

Procedure TMyApp.HandleEvent(var Event: TEvent);
Begin
  TApplication.HandleEvent(Event);
  if Event.What = evCommand then
    begin
      case Event.Command of
        cmNewDialog: NewDialog;
      else Exit;
      end;
      ClearEvent(Event);
    end;
End;

Procedure TMyApp.InitStatusLine;
var R: TRect;
Begin
  GetExtent(R);
  R.A.Y := R.B.Y - 1;
  StatusLine := New(PStatusLine, Init(R,
    NewStatusDef(0,$FFFF,
      NewStatusKey('~Alt-X~ Exit',kbAltX,cmQuit,
        NewStatusKey('~Alt-D~ Dialog',kbAltD,cmNewDialog,
          nil)),
      nil)));
End;

Procedure TMyApp.NewDialog;
Var
  Dlg : PDialog;
  R : TRect;
  IL : PInputLine;
Begin
  R.Assign(20,2,60,16);
  Dlg := New(PDialog,Init(R,'Demo Dialog'));
  with Dlg do
    begin
      R.Assign(15,5,38,6);
      IL := New(PInputLine, Init(R,60));
      Insert(IL);
      R.Assign(2,5,14,6);
      Insert(New(PLabel, Init(R,'~F~ile name : ', IL)));
    end;
  DeskTop^.ExecView(Dlg);
End;

Var
  MyApp: TMyApp;

Begin
  MyApp.Init;
  MyApp.Run;
  MyApp.Done;
End.

```

Обмен данными между строками ввода

В приведенном ниже примере показано, как производится обмен данными между двумя строками ввода. Одна строка используется для ввода информации. Затем при нажатии кнопки *Convert* происходит преобразование данных, которые отображаются в другой строке ввода.

```
{////////////////////////////////////  
INSERT.PAS: Пример переноса данных из строки ввода  
////////////////////////////////////}
```

```
uses Objects, Drivers, Views, Menus, Dialogs, App;
```

```
Const  
  cmNewDialog = 101;  
  cmConvert   = 102;
```

```
Type  
  PDataRec = ^DataRec;  
  DataRec  = Record  
    Field1: string [15];  
    Field2: string [15];  
  End;
```

```
Const  
  DemoDialogData: DataRec = (Field1: '123.45'; Field2: '');
```

```
Type  
  TMyApp = object(TApplication)  
    procedure HandleEvent(var Event: TEvent); virtual;  
    procedure InitStatusLine; virtual;  
    procedure NewDialog;  
  End;
```

```
  PDemoDialog = ^TDemoDialog;  
  TDemoDialog = object(TDialog)  
    POutput: PInputLine;  
    Pinput: PinputLine;  
    procedure HandleEvent(var Event: TEvent); virtual;  
  End;
```

```
Procedure TMyApp.HandleEvent(var Event: TEvent);  
Begin  
  TApplication.HandleEvent(Event);  
  if Event.What = evCommand then  
    begin  
      case Event.Command of  
        cmNewDialog: NewDialog;  
      else Exit;  
      end;  
      ClearEvent(Event);  
    end;  
End;
```

```
Procedure TMyApp.InitStatusLine;  
var R: TRect;
```

```

Begin
  GetExtent(R);
  R.A.Y := R.B.Y - 1;
  StatusLine := New(PStatusLine, Init(R,
    NewStatusDef(0,$FFFF,NewStatusKey('~Alt-X~ Exit',
      kbAltX,cmQuit,
      NewStatusKey('~Alt-D~ Dialog',kbAltD,cmNewDialog,
        nil))),
    nil)));
End;

```

Procedure TMyApp.NewDialog;

```

var
  Dlg      : PDemoDialog;
  R        : TRect;
  C        : Word;
  IControl : PInputLine;
  OControl : PInputLine;

Begin
  R.Assign(20,5,60,15);
  Dlg := New(PDemoDialog,Init(R,'Input/Output Demo'));
  with Dlg do
  begin
    R.Assign (12,2,37,3);
    IControl := New(PInputline, Init(R,15));
    Dlg.Insert(IControl);
    Dlg.PInput := IControl;
    R.Assign (5,2,12,3);
    Dlg.Insert(New(PLabel,Init(R,'Input',IControl))) ;
    R.Assign(12,4,37,5);
    OControl := New(PInputline,Init(R,15));
    Dlg.Insert (OControl);
    Dlg.POutput := OControl;
    R.Assign (4,4,12,5);
    Dlg.Insert(New(PLabel,Init(R,'Output',OControl))) ;
    R.Assign(7,6,18,8);
    Dlg.Insert(New(PButton,Init (R,'Convert',cmConvert,
      bfDefault))) ;
    R.Assign(20,6,30,8) ;
    Dlg.Insert(New(PButton,Init(R,'Done',cmOk,0)));
    Dlg.SelectNext(False);
    Dlg.SetData(DemodialogData);
  end;
  C := DeskTop.ExecView(Dlg);
End;

```

Procedure TDemoDialog.HandleEvent;

```

Var
  S: String;
Begin
  TDialog.HandleEvent(Event);
  if Event.What = evCommand then begin
    Case Event.Command of
      cmConvert: Begin
        S := PInput.Data;
        PInput.GetData(S);
        POutput.SetData(S);
      End;
    End;
  End;
End;

```

```

End;
end;

Var
  MyApp: TMyApp;

Begin
  MyApp.Init;
  MyApp.Run;
  MyApp.Done;
End.

```

Показанный способ может использоваться для преобразования вводимых данных, например даты, из одного формата в другой, преобразования из десятичной в шестнадцатичную систему и т.д.

Изменения в Turbo Vision 2.0

Объект TInputLine

Поле Validator

Это поле содержит указатель на объект проверки ввода, используемый совместно со строкой ввода.

Метод SetValidator

С помощью этого метода можно установить указатель на объект проверки ввода, который будет использоваться совместно с данной строкой ввода.

Вызов метода

```
Field^.SetValidator(New(PRangeValidator...
```

эквивалентен вызову:

```
Field^.Validator := New(PRangeValidator...
```

Более подробно использование объектов проверки ввода рассматривается ниже.

Расширения строки ввода

В этом разделе мы рассмотрим реализацию следующих объектов:

Объект	Назначение
TInputLineUC	Преобразует вводимые символы к символам верхнего регистра
TInputLineLC	Преобразует вводимые символы к символам нижнего регистра
TInputLinePS	Замещает вводимые символы указанным символом
TFilterInput	Производит фильтрацию вводимых символов
TInputLineNE	Эта строка ввода не может быть пустой

Объект TInputLineUC

Этот объект, являясь прямым наследником объекта *TInputLine*, преобразует вводимые символы к символам верхнего регистра.

```

PInputLineUC = ^TInputLineUC;
TInputLineUC = Object(TInputLine)
  Procedure HandleEvent(Var Event : TEvent);   Virtual;
End;

Procedure TInputLineUC.HandleEvent(Var Event: TEvent);
Begin
  If ((Event.What = evKeyDown) AND (Event.CharCode IN ['a'..'z']))
  Then Event.CharCode := CHR((ORD(Event.CharCode) - 32));
  Inherited HandleEvent(Event);
End;
```

Как видно из реализации этого объекта, мы переопределяем метод обработки событий *HandleEvent* и при наступлении события "нажатие клавиши" проверяем, попадает ли введенный с помощью клавиши символ в диапазон символов, которые преобразуются. Такими символами в нашем случае являются латинские символы от 'a' до 'z' и русские символы от 'а' до 'я'. В случае латинских символов мы уменьшаем код символа на 32 (что дает нам символ верхнего регистра). В случае с русскими символами мы имеем дело с двумя дополнительными диапазонами. Первый из них - символы от 'а' до 'п' обрабатываются точно так же, как и латинские, а для символов от 'р' до 'я' коды символов уменьшаются на 80. Обработчик событий, поддерживающий преобразование как латинских, так и русских символов, будет выглядеть следующим образом:

```

Procedure TInputLineUC.HandleEvent(Var Event: TEvent);
Begin
  If Event.What = evKeyDown Then
    Begin
      {a-z}
      If Event.CharCode IN ['a'..'z']
        Then Event.CharCode := CHR((ORD(Event.CharCode) - 32));
      {a-п}
      If Event.CharCode IN ['a'..'п']
        Then Event.CharCode := CHR((ORD(Event.CharCode) - 32));
      {p-я}
      If Event.CharCode IN ['p'..'я']
        Then Event.CharCode := CHR((ORD(Event.CharCode) - 80));
    End;
    Inherited HandleEvent(Event);
End;

```

Объект TInputLineLC

Аналогичным образом может быть реализован объект, преобразующий вводимые символы в символы нижнего регистра:

```

PInputLineLC = ^TInputLineLC;
TInputLineLC = Object(TInputLine)
  Procedure HandleEvent(Var Event : TEvent); virtual;
End;

Procedure TInputLineLC.HandleEvent(Var Event: TEvent);
Begin
  If Event.What = evKeyDown Then
    Begin
      {a-z}
      If Event.CharCode IN ['a'..'z']
        Then Event.CharCode := CHR((ORD(Event.CharCode) + 32));
      {a-п}
      If Event.CharCode IN ['a'..'п']
        Then Event.CharCode := CHR((ORD(Event.CharCode) + 32));
      {p-я}
      If Event.CharCode IN ['p'..'я']
        Then Event.CharCode := CHR((ORD(Event.CharCode) + 80));
    End;
    Inherited HandleEvent(Event);
End;

```

Объект TInputLinePS

Следующий объект, который построен на основе объекта *TInputLine*, - объект *TInputLinePS*. Это объект, обладающий всеми свойствами строки ввода, но не отображающий вводимые пользователем символы, - все

символы отображаются как '' или любой другой заданный в конструкторе символ.

Такой объект может быть использован для ввода пароля.

```
InputLinePS = ^TInputLineNE;  
TInputLinePS = Object(TInputLine)  
NEChar : Char; {символ для отображения}  
Constructor Init(Var Bounds: TRect; AMaxLen: Integer;  
                 Ch : Char);  
Procedure Draw; virtual;  
End;
```

В отличие от рассмотренных нами ранее объектов, реализация объекта *TInputLinePS* требует более комплексного подхода. Сначала переопределим конструктор и введем новый параметр, позволяющий задавать символ, который будет отображаться вместо вводимого.

```
Constructor TInputLinePS.Init;  
Begin  
  TInputLine.Init(Bounds,AMaxLen);  
  NEChar := Ch;  
End;
```

Затем реализуем функцию *FillString*, которая возвращает строку указанной длины, заполненную определенным символом. Назначение этой функции станет понятнее чуть позже.

```
Function FillString(Len : Byte; Ch : Char) : String;  
Var  
  S : String;  
Begin  
  If (Len > 0) Then  
    Begin  
      S[0] := Chr(Len);  
      FillChar(S[1], Len, Ch);  
      FillString := S;  
    End  
  Else FillString := '';  
End;
```

И наконец, переопределим метод *Draw*. Этот метод вызывается методом *HandleEvent* для отображения введенных символов.

```
Procedure TInputLinePS.Draw;  
Var  
  Original : String;  
Begin  
  If (Length(Data) > 0) Then  
    Begin
```

```

Original := Data^;
{Заполнить буфер определенным символом}
Data^ := FillString (Length(Data^), NEChar);
Inherited Draw;
Data^ := Original;
End
Else Inherited Draw;
End;

```

Таким образом, для того, чтобы любой введенный в строке ввода символ отображался одним и тем же символом, нам необходимо сохранить оригинальную информацию, заменить ее на строку, заполненную определенным символом, отобразить эту строку (с помощью метода *TInputLine.Draw*), а затем восстановить оригинальную информацию. В результате введенная информация не будет отображена, но попадет по назначению в неискаженном виде. Отметим, что для создания "полной" версии объекта *TInputLinePS*, нам необходимо реализовать еще два метода: *Load* и *Store*, так как этот объект, в отличие от оригинального, имеет дополнительное поле.

```

Constructor TInputLinePS.Load(var S: TStream);
Begin
  Inherited Load(S);
  S.Read(NEChar, SizeOf(Char));    {Записать дополнительное поле}
End;

Procedure TInputLinePS.Store(var S: TStream);
Begin
  Inherited Store(S);
  S.Write(NEChar, SizeOf(Char));    {Считать дополнительное поле}
End;

```

Для использования объекта *TInputLinePS* в потоках, нам также необходимо реализовать регистрационную запись:

```

RInputLinePS : TStreamRec = (
  ObjType      : 700;
  VmtLink      : OfS(Kind(TInputLinePS));
  Load         : @TInputLinePS.Load;
  Store        : @TInputLinePS.Store);

```

Напомним, что перед использованием объекта в потоках, его необходимо зарегистрировать:

```
RegisterType(RInputLinePS)
```

Следующий объект, который мы рассмотрим, предназначен для фильтрации вводимой информации. В конструкторе этого объекта задается набор символов, допустимых для ввода, остальные символы игнорируются.

Создадим новый тип данных:

```
TCharSet = Set of Char;
```

В объекте *TFilterInput* переопределены два метода: конструктор *Init* и метод обработки событий *HandleEvent*.

```
PFilterInput = ^TFilterInput;  
TFilterInput = Object(TInputLine)  
  CharsAllowed : TCharSet;  
  Constructor Init(var Bounds : TRect; AMaxLen : Integer;  
                  Allowed : CharSet);  
  Procedure HandleEvent(var Event : TEvent);      virtual;  
  End;
```

Конструктор отличается от оригинального только наличием дополнительного параметра, задающего набор допустимых символов:

```
Constructor TFilterInput.Init;  
Begin  
  Inherited Init(Bounds, AMaxLen);  
  CharsAllowed := Allowed;  
End;
```

Обработчик событий реагирует на событие "нажатие клавиши" и проверяет, находится ли введенный символ в наборе допустимых символов. Если это не так, то символ игнорируется. Остальные события обрабатываются методом *HandleEvent* объекта *TInputLine*.

```
Procedure TFilterInput.HandleEvent;  
Begin  
  If (Event.What = evKeyDown) Then  
  Begin  
    If Event.CharCode <> #0 Then      {Введен символ}  
    Begin  
      If Event.CharCode In [#1..#$1B] Then Exit;  
      If NOT(Event.CharCode In CharsAllowed)  
      Then ClearEvent(Event);  
    End;  
  End;  
  TInputLine.HandleEvent(Event);  
End;
```

Используя этот объект, можно фильтровать вводимую информацию. Например, для разрешения ввода только цифр конструктор объекта *TFilterInput* вызывается следующим образом:

```
Filter := New(PFilterInput, Init(R, 10, ['0'..'9']))
```

Для ввода только символов верхнего регистра:

```
Filter := New(PFilterInput, Init(R, 10, ['A'..'Z', 'A'..'Я']))
```

Таким образом, допустимы различные комбинации фильтруемых символов, что позволяет использовать этот объект в различных приложениях.

Завершая рассмотрение объекта *TFilterInput*, отметим, что приведенная ниже модификация делает этот объект более гибким.

Реализуем еще один метод - *SetChars*, который будет использоваться для задания набора допустимых символов:

```
TFilterInput.SetChars(Allowed : TCharSet);  
Begin  
  CharsAllowed := Allowed;  
End;
```

В этом случае конструктор будет выглядеть следующим образом:

```
Constructor TFilterInput.Init;  
Begin  
  Inherited Init(Bounds, AMaxLen);  
  { Разрешить ввод всех символов }  
  CharsAllowed := [#0..#255];  
End;
```

Таким образом, при вызове конструктора, все символы допустимы ко вводу, а с помощью метода *SetChars* набор допустимых символов можно изменять динамически.

Измененное объявление объекта *TFilterInput* будет выглядеть следующим образом:

```
PFilterInput = ^TFilterInput;  
TFilterInput = Object(TInputLine)  
  CharsAllowed : TCharSet;  
  Constructor Init(var Bounds : TRect; AMaxLen : Integer);  
  Procedure SetChars(Allowed : TCharSet);  
  Procedure HandleEvent(var Event : TEvent); virtual;  
End;
```

Также отметим, что для использования этого объекта в потоках необходимо реализовать методы *Load* и *Store* для

сохранения набора допустимых символов и регистрационную запись *RFilterInput*.

Объект *TInputLineNE*

Следующий объект, который мы рассмотрим, - это строка ввода, которая не может быть пустой. Использование такого объекта может быть полезным в тех случаях, когда требуемая от пользователя информация не может не быть введена.

```
InputLineNE = TInputLineNE;  
TInputLineNE = Object(TInputLine)  
Function Valid(Command : Word) : Boolean;      virtual;  
End;
```

Как видно из объявления объекта, мы переопределяем метод *Valid* стандартного объекта *TInputLine*:

```
Function TInputLineNE.Valid;  
Begin  
  If (NOT (Command IN [cmValid, cmCancel])) AND  
    (Data = "") Then  
    Begin  
      Select;  
      MessageBox('Это поле должно быть заполнено',  
                  Nil, mfError + mfOkButton);  
      Valid := False;  
    End  
  Else  
    Valid := Inherited Valid(Command);  
End;
```

В переопределенном методе мы проверяем, содержит ли поле *Data* данные или нет. Если это поле пустое, мы устанавливаем фокус на это поле и сообщаем пользователю об ошибке.

Объект *TScrollBar*

Объект *TScrollBar* предназначен для создания полос прокрутки, использующие совместно со списками, текстовыми редакторами и различными другими отображаемыми объектами, которые могут осуществлять просмотр текста. Когда объекты типа *TScrollBar* используются совместно с объектами *TListBox* или *TListViewer*, последние автоматически обновляют поля

объекта *TScrollBar*. В результате при использовании объекта *TScrollBar* вместе со списком все, что необходимо сделать - это вызвать конструктор *Init*. Также при изменении положения бегунка полосы прокрутки объект *TScrollBar* использует функцию *Message* для уведомления владельца произошедших изменений.

Пример использования полосы прокрутки совместно с объектом *TListBox* приведен ниже.

```
{//////////////////////////////////////
LISTBOX.PAS: Использование объекта TListBox
//////////////////////////////////////}
uses Objects, App, Drivers, Dialogs, Views, Menus, MsgBox;
Const
  cmDialog    = 1000;
  cmListBox   = 1001;
Type
  PSCollection = ^TSCollection;
  TSCollection = Object(TSortedCollection)
    function Compare(Key1,Key2 : Pointer) : Integer; Virtual;
End;

TDemoApp = Object(TApplication)
  procedure InitStatusLine;                virtual;
  procedure HandleEvent(var Event : TEvent); virtual;
  function  BuildList : PSCollection;
  procedure DemoDialog;
End;

PDialogBox = ^TDialog;
TDialogBox = Object(TDialog)
  procedure HandleEvent(var Event : TEvent); virtual;
End;

PMLListBox = ^TMLListBox;
TMLListBox = Object(TListBox)
  procedure HandleEvent(var Event : TEvent); virtual;
End;

TListBoxRec = Record
  List      : PCollection;
  Selection : Word;
End;

Var
  ListBoxRec : TListBoxRec;
  LB         : PMLListBox;

Function TSCollection.Compare;
Begin
  If PString(Key1)^ < PString(Key2)^ Then Compare := -1
  else
  If PString(Key1)^ > PString(Key2)^ Then Compare := 1
  else
    Compare := 0;
End;
```

```

Procedure TMLListBox.HandleEvent;
Begin
  If ((Event.What = evKeyDown) AND (Event.KeyCode = kbEnter))
  OR ((Event.What = evMouseDown) AND (Event.Double)) Then
    Begin
      MessageBox('C'You selected : ' + GetText(Focused,128),Nil, mfOkButton);
    End
  Else Inherited HandleEvent(Event);
End;

```

```

Procedure TDemoApp.InitStatusLine;
Var
  R : TRect;
Begin
  GetExtent(R);
  R.A.Y := R.B.Y - 1;
  StatusLine := New(PStatusLine, Init(R,
    NewStatusDef(0, $FFFF,
      NewStatusKey('~Alt-X~ Exit', kbAltX, cmQuit,
        NewStatusKey('~Alt-D~ Dialog', kbAltD, cmDialog,
          Nil)),
    Nil)
  ));
End;
Function TDemoApp.BuildList;
Var
  List : PSCollection;
Begin
  List := New(PSCollection, Init(10,10));
  With List do
    Begin
      Insert(NewStr('Turbo Pascal'));
      Insert(NewStr('Borland C + +'));
      Insert(NewStr('Object Vision'));
      Insert(NewStr('Paradox'));
      Insert(NewStr('Quattro Pro'));
      Insert(NewStr('Paradox Engine'));
      Insert(NewStr('Paradox for Window'));
      Insert(NewStr('SideKick'));
      Insert(NewStr('InterBase'));
      Insert(NewStr('Quattro Pro for Windows'));
      Insert(NewStr('dBase 2.5'));
    End;
  BuildList := List;
End;

```

```

Procedure TDemoApp.HandleEvent;
Begin
  Inherited HandleEvent(Event);
  if Event.What = evCommand then
    begin
      case Event.Command of
        cmDialog : DemoDialog;
      else
        Exit;
      end;
      ClearEvent(Event);
    end;
End;

```

```

Procedure TDemoApp.DemoDialog;

Var
  Dialog : PDialogBox;
  R      : TRect;
  SB     : PScrollBar;
Begin
  R.Assign(20,5,60,20);
  Dialog := New(PDialogBox,Init(R,'Demo Dialog'));
  With Dialog^ do

Begin
  R.Assign(10,2,30,3);
  Insert(New(PStaticText, Init(R, 'C'Borland Products')));
  R.Assign(36,3,37,13);
  SB := New(PScrollBar,Init(R));
  Insert(SB);
  R.Assign(5,3,35,13);
  LB := New(PMListBox,Init(R,1,SB));
  ListBoxRec.List := BuildList;
  ListBoxRec.Selection := 0;
  LB^.SetData(ListBoxRec);
  LB^.Options := LB^.Options OR ofFramed;
  Insert(LB);
  Palette := dpCyanDialog;
End;

ExecuteDialog(Dialog,Nil);
End;

Procedure TDialogBox.HandleEvent;
Begin
  Inherited HandleEvent(Event);
End;

Var
  DemoApp : TDemoApp;

Begin
  DemoApp.Init;
  DemoApp.Run;
  DemoApp.Done;
End.

```

Обратите внимание на реализацию метода *TDemoApp.DemoDialog*. В ней создается экземпляр объекта *TScrollBar*, который используется совместно со списком. Необходимым условием нормального функционирования полосы прокрутки является правильное указание координат и использование метода *Insert* для отображения полосы прокрутки внутри панели диалога. В конструкторе списка необходимо указать параметр, соответствующий указателю на полосу прокрутки.

Обычно полосы прокрутки используются совместно с окнами или списками для перемещения внутри них. Полоса прокрутки также может использоваться для ввода информации, что показано на приведенном примере.

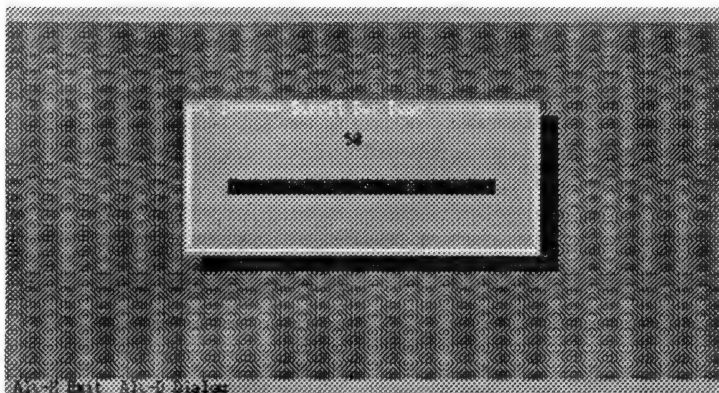


Рис.3.6.Использование объекта TScrollBar

```
{//////////////////////////////////////
SB_DEMO.PAS: Пример использования полосы прокрутки
//////////////////////////////////////}
uses Objects, Drivers, Views, Menus, Dialogs, App, MsgBox;

Const
  cmNewDialog = 101;

Type
  TMyApp = object(TApplication)
    procedure HandleEvent(var Event: TEvent); virtual;
    procedure InitStatusLine; virtual;
    procedure NewDialog;
  End;

  PDemoDialog = ^TDemoDialog;
  TDemoDialog = object(TDialog)
    Value : PStaticText;
    SB : PScrollBar;
    procedure HandleEvent(var Event:TEvent); virtual;
  End;

Procedure TMyApp.HandleEvent(var Event: TEvent);
Begin
  TApplication.HandleEvent(Event);
  if Event.What = evCommand then
```

```

begin
  case Event.Command of
    cmNewDialog: NewDialog;
  else Exit;
  end;
  ClearEvent(Event);
end;
End;

Procedure TMyApp.InitStatusLine;
var R: TRect;
Begin
  GetExtent(R);
  R.A.Y := R.B.Y - 1;
  StatusLine := New(PStatusLine, Init(R,
    NewStatusDef(0,$FFFF,
      NewStatusKey('~Alt-X~ Exit',kbAltX,cmQuit,
        NewStatusKey('~Alt-D~ Dialog',kbAltD,cmNewDialog,
          nil)),
      nil)));
End;

```

```

Procedure TMyApp.NewDialog;
Var
  Dlg : PDemoDialog;
  R : TRect;
Begin
  R.Assign(20,5,60,15);
  Dlg := New(PDemoDialog,Init(R,'Scroll Bar Demo'));
  with Dlg do
    begin
      R.Assign(5,5,35,6);
      SB := New(PScrollBar,Init(R));
      SB.SetParams(0,0,100,10,10);
      Insert(SB);
      R.Assign(30,2,33,3);
      Value := New(PStaticText,Init(R,'0'));
      Value.Options := Value.Options OR ofCenterX;
      Insert(Value);
    end;
  DeskTop.ExecView(Dlg);
End;

```

```

Procedure TDemoDialog.HandleEvent;
Var
  S : String;
Begin
  TDialog.HandleEvent(Event);
  If Event.What = evBroadcast then
    Case Event.Command of
      cmScrollBarChanged :
        Begin
          Str(SB.Value,S);
          Value.Text := S;
          Value.Draw;
        End;
      else
        Exit;
      End;
    ClearEvent(Event);
  End;

```

```

End;

Var
  MyApp: TMyApp;

Begin
  MyApp.Init;
  MyApp.Run;
  MyApp.Done;
End.

```

В методе *NewDialog* создается панель диалога, в которую помещается горизонтальная полоса прокрутки и статический текст. Для полосы устанавливается диапазон 100 и шаг бегунка 10:

```

SB := New(PScrollBar, Init(R));
SB^.SetParams(0, 0, 100, 10, 10);

```

В обработчике событий проверяется сообщение *cmScrollBarChanged* - изменение положения бегунка - и отображается его текущее положение.

```

If Event.What = evBroadcast then
  Case Event.Command of
    cmScrollBarChanged :
      Begin
        Str(SB^.Value, S);
        Value^.Text := S;
        Value^.Draw;
      End;

```

Перемещаемые элементы управления

Реализовать свойство перемещения любого элемента управления довольно просто: достаточно переопределить метод. Предположим, мы выбрали правую кнопку мыши в качестве средства для перемещения элемента управления. В приведенном ниже примере показано как переопределить методы *HandleEvent* кнопки, строки редактирования и списка.

```

{//////////////////////////////////////
TVDRAG.PAS: Пример реализации перемещаемых элементов
управления
////////////////////////////////////}
uses Dos, Objects, App, Dialogs, Views, Menus, Drivers, MsgBox;

Const
  cmNewDlg = 100;

```

```

Type
  PMyInputLine = ^TMyInputLine;
  TMyInputLine = Object(TInputLine)
    procedure HandleEvent(var Event : TEvent);   virtual;
    procedure WriteSource;
  End;
  PMyButton    = ^TMyButton;
  TMyButton    = Object(TButton)
    procedure HandleEvent(var Event : TEvent);   virtual;
  End;

  PMyListBox   = ^TMyListBox;
  TMyListBox   = Object(TListBox)
    procedure HandleEvent(var Event : TEvent);   virtual;
  End;

  TMyApp = Object(TApplication)
    Dialog : PDialog;
    constructor Init;
    procedure InitStatusLine;                   virtual;
    procedure HandleEvent(var Event : TEvent);   virtual;
    function  GetPalette : PPalette;             virtual;
    procedure NewDlg;
  End;

Procedure TMyInputLine.HandleEvent;
Var
  R      : TRect;
  Min,Max : TPoint;
Begin
  If Event.What AND evMouseDown = evMouseDown then
    Begin
      Owner^.GetExtent(R);
      R.Grow(-1, -1);
      SizeLimits(Min, Max);
      case Event.Buttons of
        mbRightButton:
          begin
            DragView(Event, dmDragMove or DragMode, R, Min, Max);
            ClearEvent(Event);
          end;
      end;
    end;
  Inherited HandleEvent(Event);
End;

Procedure TMyButton.HandleEvent;
Var
  R      : TRect;
  Min,Max : TPoint;
Begin
  If Event.What AND evMouseDown = evMouseDown then
    Begin
      Owner^.GetExtent(R);
      R.Grow(-1, -1);
      SizeLimits(Min, Max);
      case Event.Buttons of
        mbRightButton:
          begin

```

```

        DragView(Event, dmDragMove or DragMode, R, Min, Max);
        ClearEvent(Event);
    end;
end;
Inherited HandleEvent(Event);
End;

Procedure TMyListBox.HandleEvent;
Var
    R      : TRect;
    Min,Max : TPoint;
Begin
    If Event.What AND evMouseDown = evMouseDown then
        Begin
            Owner^.GetExtent(R);
            R.Grow(-1, -1);
            SizeLimits(Min, Max);
            case Event.Buttons of
                mbRightButton:
                    begin
                        DragView(Event, dmDragMove or DragMode, R, Min, Max);
                        ClearEvent(Event);
                    end;
            end;
        end;
    Inherited HandleEvent(Event);
End;

Constructor TMyApp.Init;
Var
    R : TRect;
Begin
    Inherited Init;
    with Desktop^ do
        Begin
            Background^.GetExtent(R);
            Delete(Background);
            Dispose(Background,Done);
            R.Grow(0,1);
            Background := New(PBackground,Init(R,'_'));
            Insert(Background);
        end;
End;

Function TMyApp.GetPalette;
Const
    MyColor : TPalette = CColor;
Begin
    MyColor[1] := #F0F;
    GetPalette := @MyColor;
End;

Procedure TMyApp.InitStatusLine;
Var
    R : TRect;
Begin
    GetExtent(R);
    R.A.Y := R.B.Y - 1;
    StatusLine := New(PStatusLine,Init(R,
        NewStatusDef(0,$FFFF,
            NewStatusKey('X~ Exit',kbAltX,cmQuit,

```

```

    NewStatusKey('~Alt-D~ Dialog', kbAltD, cmNewDlg,
    nil)),
    nil)
  ));
End;

```

```

Procedure TMyApp.NewDlg;

```

```

Var
  R      : TRect;
  Input  : PMyInputLine;
  LBox   : PMyListBox;
  Control : Word;
Begin
  R.Assign(20,5,60,20);
  Dialog := New(PDialog, Init(R, 'Drag Controls'));
  With Dialog^ do
    Begin
      R.Assign(2,3,38,4);
      Input := New(PMyInputLine, Init(R, 40)); Insert(Input);
      R.Assign(2,5,38,6);
      Input := New(PMyInputLine, Init(R, 40)); Insert(Input);
      R.Assign(10,7,30,10);
      LBox := New(PMyListBox, Init(R, 1, Nil));
      Insert(LBox);

      R.Assign(10,11,20,13);
      Insert(New(PMyButton, Init(R, '~O~k', cmOk, bfDefault)));
      R.Assign(20,11,30,13);
      Insert(New(PMyButton, Init(R, 'Cancel', cmCancel, bfNormal)));
      SelectNext(False);
    End;
  Dialog^.Flags := Dialog^.Flags OR wfGrow OR wfZoom;
  Control := ExecView(Dialog);
End;

```

```

Procedure TMyApp.HandleEvent;

```

```

Begin
  inherited HandleEvent(Event);
  if Event.What = evCommand Then
    Begin
      Case Event.Command of
        cmNewDlg : NewDlg;
      else Exit;
      End;
    End;
  ClearEvent(Event);
End;

```

```

Var
  MyApp : TMyApp;

```

```

Begin
  MyApp.Init;
  MyApp.Run;
  MyApp.Done;
End.

```

Таким образом, используя перемещаемые элементы управления, мы можем позволить пользователю настроить интерфейс более естественным для него образом. Такая техника может быть использована и при реализации различных средств автоматизации создания программ.

Отображение статуса операции

При выполнении операций, которые могут занимать длительное время (например, вывод на принтер или сортировка базы данных) бывает удобным информирование пользователя о ходе операции, а также предоставление возможности аварийного завершения такой операции. Как это сделать, показано в приведенном ниже примере.

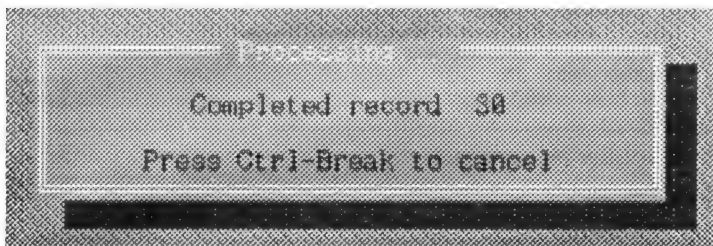


Рис.3.7. Отображение статуса операции

```
{//////////////////////////////////////
STATBOX.PAS - Пример создания информационной панели
//////////////////////////////////////}
uses Crt, Objects, Drivers, Views, Dialogs, App;

Type
  PDemo = ^TDemo;
  TDemo = Object (TApplication)
    constructor Init;
  End;

  PStatusDialog = ^TStatusDialog;
  TStatusDialog = Object (TDialog)
    Message, Value: PStaticText;
    constructor Init;
    procedure Update (Status: Word; AValue: Word); virtual;
  End;

Constructor TDemo.Init;
Var
  D : PStatusDialog;
  I : Integer;
  E : TEvent;
```

```

Begin
  Inherited Init;
  D := New (PStatusDialog, Init);
  Desktop^.Insert (D);
  for I := 1 to 10 do
    begin
      D^.Update (cmValid, I * 10);
      if CtrlBreakHit then
        begin
          CtrlBreakHit := False;
          GetEvent (E);
          D^.Update (cmCancel, I * 10);
          repeat GetEvent (E) until (E.What = evKeyDown)
            or (E.What = evMouseDown);
          Desktop^.Delete (D);
          Dispose (D, Done);
          Exit;
        end;
      Delay (1000);
    end;
  D^.Update (cmOK, 100);
  repeat GetEvent (E) until (E.What = evKeyDown)
    or (E.What = evMouseDown);
  Desktop^.Delete (D);
  Dispose (D, Done);
end;

Constructor TStatusDialog.Init;
Var
  R: TRect;
Begin
  R.Assign (20, 6, 60, 12);
  Inherited Init(R, 'Processing...');
  Flags := Flags and not wfClose;
  R.Assign (10, 2, 30, 3);
  Insert (New (PStaticText, Init (R, 'Completed record xxx')));
  R.Assign (27, 2, 30, 3);
  Value := New (PStaticText, Init (R, ' 0'));
  Insert (Value);
  R.Assign (2, 4, 38, 5);
  Message := New (PStaticText, Init (R,
    ' Press Ctrl-Break to cancel '));
  Insert (Message);
End;

Procedure TStatusDialog.Update (Status: Word; AValue: Word);
Var
  ValStr: String[3];

Begin
  case Status of
    cmCancel: begin
      DisposeStr (Message^.Text);
      Message^.Text := NewStr (' Cancelled - press any key ');
      Message^.DrawView;
    end;
    cmOK: begin
      DisposeStr (Message^.Text);
      Message^.Text := NewStr (' Completed - press any key ');
    end;
  end;
end;

```

```

    Message^.DrawView;
end;
end;
Str (AValue:3,ValStr);
DisposeStr (Value^.Text);
Value^.Text := NewStr (ValStr);
Value^.DrawView;
End;

```

```

Var
Demo: TDemo;

```

```

Begin
Demo.Init;
Demo.Run;
Demo.Done;
End.

```

Отображение разноцветного текста

Ниже на примере показано, как создать объект - наследник *TStaticText*, который может выводить текст указанным цветом.

```

Type
PColoredText = ^TColoredText;
TColoredText = Object(TStaticText)
    Attr : Byte;
    constructor Init(var Bounds: TRect; AText: String; Attribute : Byte);
    constructor Load(var S: TStream);
    function GetTheColor : byte; virtual;
    procedure Draw; virtual;
    procedure Store(var S: TStream);
end;

```

```

const
RColoredText: TStreamRec = (
    ObjType: 611;
    VmtLink: Obj(TypeOf(TColoredText)^);
    Load: @TColoredText.Load;
    Store: @TColoredText.Store
);

```

```

Constructor TColoredText.Init(var Bounds: TRect; AText: String; Attribute :
Byte);

```

```

Begin
TStaticText.Init(Bounds, AText);
Attr := Attribute;
End;

```

```

Constructor TColoredText.Load(var S: TStream);
Begin
TStaticText.Load(S);
S.Read(Attr, Sizeof(Attr));
End;

```

```

Procedure TColoredText.Store(var S: TStream);
Begin
  TStaticText.Store(S);
  S.Write(Attr, Sizeof(Attr));
End;

Function TColoredText.GetTheColor : byte;
Begin
  if AppPalette = apColor then
    GetTheColor := Attr
  else
    GetTheColor := GetColor(1);
End;

Procedure TColoredText.Draw;
Begin
{
  Если у вас есть исходный текст Turbo Vision, то
  скопируйте метод TStaticText.Draw из файла DIALOGS.PAS.
  Измените первую строку:
  вместо Color := GetColor(1);
  напишите Color := GetTheColor;
}
End;

```

Панель диалога с возможностью прокрутки информации

Часто при разработке приложений мы сталкиваемся с проблемой нехватки места для расположения всех интерфейсных элементов на экране. Одно из решений этой проблемы предлагается ниже. Показано, как реализовать панель диалога с возможностью прокрутки информации. В стандартную панель диалога добавляется полоса прокрутки, с помощью которой происходит прокрутка вложенных интерфейсных элементов. В данном примере показано, как осуществить вертикальную прокрутку. По тому же принципу можно осуществить и горизонтальную прокрутку, реализацию которой я оставляю читателю в качестве упражнения. Прокрутка осуществляется процедурой *ScrollSub View*, которая вызывается итератором *ForEach* для каждого вложенного интерфейсного элемента.

В конструкторе объекта *TScrollDialog*, помимо размера и заголовка, задаются следующие параметры:

VSize	размер по вертикали
Step	шаг прокрутки
PgStep	шаг полосы прокрутки

unit ScrDlg;

```
{////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////}
SCRLDLG - Реализация объекта TScrollDialog - панели диалога с
возможностью прокрутки
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////}
```

```
Interface
uses Objects, Drivers, Views, Dialogs;
Type
PScrollDialog = ^TScrollDialog;
TScrollDialog = Object(TDialog)
  LastValue : Integer;
  ScrollBar : PScrollBar;
  constructor Init(var R : TRect; ATitle : TTitleStr;
    VSize, Step, PgStep : Integer);
  procedure HandleEvent(var Event : TEvent); virtual;
  procedure SetState(AState : Word; Enable : Boolean); virtual;
End;
```

Implementation

```
Constructor TScrollDialog.Init;
Begin
  Inherited Init(R, ATitle);
  ScrollBar := StandardScrollBar(sbVertical or sbHandleKeyboard);
  ScrollBar^.SetRange(0, VSize-Size.Y);
  ScrollBar^.SetStep(PgStep, Step);
End;
```

```
Procedure TScrollDialog.HandleEvent;
Var
  Step : Integer;
```

```
Procedure ScrollSubView(P : PView); FAR;
Begin
  If (PFrame(P) <> Frame) AND (PScrollBar(P) <> ScrollBar) Then
  Begin
    P^.MoveTo(P^.Origin.X, P^.Origin.Y-Step);
    if (P^.Origin.Y < 1) OR (P^.Origin.Y+P^.Size.Y > Size.Y-1)
    then P^.Hide
    else P^.Show;
  End;
End;
Begin
  if (Event.What = evBroadcast) AND
    (Event.Command = cmScrollBarChanged) then

  Begin
    Step := ScrollBar^.Value - LastValue;
    LastValue := ScrollBar^.Value;
    Lock;
    ForEach(@ScrollSubView);
    Unlock;
  End;
  Inherited HandleEvent(Event);
End;
```

```

Procedure TScrollDialog.SetState;
Begin
  if ((AState and sfModal) = sfModal) AND Enable then
    Message(@Self, evBroadcast, cmScrollBarChanged, ScrollBar);
  Inherited SetState(AState, Enable);
End;
End.

```

После того как панель диалога реализована, проверим ее в работе: в приведенной ниже программе создается 14 строк ввода (объект *TInputLine*) и 14 меток (объект *TStaticText*), которые могут отображаться в панели диалога в зависимости от положения бегунка полосы прокрутки.

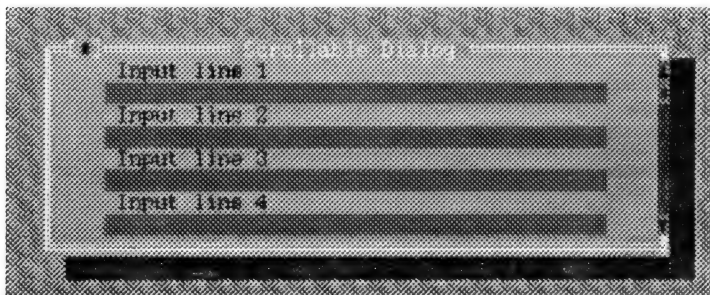


Рис.3.8.Панель диалога со скроллингом

```

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
SD_DEMO.PAS - Пример использования объекта TScrollDialog
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

```

```

uses
  Objects, App, Menus, Drivers, Views, Dialogs, ScrlDlg;
Const
  cmDemo = 1000;
Type
  TDemoApp = Object(TApplication)
    procedure InitMenuBar;          virtual;
    procedure Demo;
    procedure HandleEvent(var Event : TEvent);  virtual;
  End;

Procedure TDemoApp.InitMenuBar;
Var
  R : TRect;
Begin
  GetExtent(R);
  R.B.Y := R.A.Y + 1;
  MenuBar := New(PMenuBar, Init(R, NewMenu(
    NewSubMenu('D`emo', hcNoContext, NewMenu(

```

```

NewItem('D~ialog',",kbNoKey, cmDemo, 0,
NewLine(
NewItem('Q~uit','Alt-Q',kbAltQ, cmQuit, hcNoContext,
Nil))),
Nil))));
End;

Procedure TDemoApp.Demo;
Var
  R      : TRect;
  Dialog : PScrollDialog;
  C      : PStaticText;
  InputLine : PInputLine;
  I      : Byte;
  S      : String[2];
  B      : PButton;
Begin
  R.Assign(15,5,65,15);
  Dialog := New(PScrollDialog, Init(R,'Scrollable Dialog', 30,1,6));
  With Dialog ^ do
  Begin
    For I := 0 to 13 do
    Begin
      R.Assign(5,2+I*2,45,3+I*2);
      InputLine := New(PInputLine,Init(R,50));
      Insert(InputLine);
      R.Assign(6,1+I*2,46,2+I*2);
      Str(I+1,S);
      C := New(PStaticText,Init(R,'Input line ' + S));
      Insert(C);
    End;
  End;
  Desktop ^ .ExecView(Dialog);
  Dispose(Dialog, Done);
End;

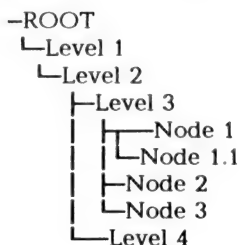
Procedure TDemoApp.HandleEvent;
Begin
  Inherited HandleEvent(Event);
  If Event.What = evCommand Then
  Begin
    Case Event.Command of
      cmDemo : Demo;
    Else Exit;
  End;
  ClearEvent(Event);
End;

End;

Var
  DemoApp : TDemoApp;
Begin
  DemoApp.Init;
  DemoApp.Run;
  DemoApp.Done;
End.

```

Два объекта, находящиеся в этом модуле, - *TOutline* и *TOutlineViewer* - предназначены для управления отображением иерархий различной вложенности. Например, эти объекты могут использоваться для отображения структуры каталога на диске, иерархий объектов и т.п. В приведенном ниже примере показано отображение следующей иерархии:



Для отображения такой иерархии может быть использована следующая программа:

```

{-----
Пример использования объекта TOutline
-----}

```

```

uses Objects,App,Dialogs,Views,Drivers,Outline;

```

```

Type

```

```

  POutDlg = ^TOutDlg;

```

```

  TOutDlg = Object(TWindow)

```

```

    constructor Init(MainTree : PNode);

```

```

  End;

```

```

  TMyApp = Object(TApplication)

```

```

    Tree : PNode;

```

```

    constructor Init;

```

```

  End;

```

```

Function BuildTree : PNode;

```

```

Begin

```

```

  BuildTree := NewNode('ROOT',

```

```

    NewNode('Level 1',

```

```

        NewNode('Level 2',

```

```

            NewNode('Level 3',NewNode('Node 1',

```

```

                NewNode('Node 1.1',Nil,Nil),

```

```

                NewNode('Node 2',Nil,

```

```

                    NewNode('Node 3',Nil,Nil))),

```

```

                NewNode('Level 4',Nil,Nil)),

```

```

                Nil), Nil), Nil);

```

```

End;

```

```

Constructor TOutDlg.Init;
Var
  R      : TRect;
  HScrollBar : PScrollBar;
  VScrollBar : PScrollBar;
  Outline  : POutline;
Begin
  R.Assign(0,0,50,20);
  inherited Init(R,'Outline Demo',wnNoNumber);
  Options := Options or ofCentered;
  VScrollBar := StandardScrollBar(sbVertical or sbHandleKeyboard);
  HScrollBar := StandardScrollBar(sbHorizontal or sbHandleKeyboard);
  Insert(VScrollBar);Insert(HScrollBar);
  R.Grow(-1,-1);
  Outline := New(POutline,Init(R,HScrollBar,VScrollBar,MainTree));
  Insert(Outline);
End;

Constructor TMyApp.Init;
Begin
  inherited Init;
  Tree := BuildTree;
  InsertWindow(New(POutDlg,Init(Tree)));
End;

Var MyApp : TMyApp;

Begin
  MyApp.Init;
  MyApp.Run;
  MyApp.Done;
End.

```

При создании объекта типа *TOutline* указываются две полосы прокрутки (горизонтальная и вертикальная), с помощью которых возможно перемещение изображения на экране.

Модуль DIALOGS

Модуль *Dialogs* содержит определения следующих команд, которые используются отображаемыми объектами:

Команда	Значение	Описание
cmRecordHistory	60	\ Эти команды используются рядом объектов /
cmGrabDefault	61	
cmReleaseDefault	62	

Команды, определенные в модуле *Dialogs*, используются рядом отображаемых объектов и не представляют практического интереса для разработчиков.

Модуль STDDLГ

В модуле *StdDlg* определен ряд команд, которые используются панелями диалога, реализованными в этом модуле.

Команда	Значение	Описание	
cmFileOpen	800	Нажата	кнопка
		Open	
cmFileReplace	801	Нажата	кнопка
		Replace	
cmFileClear	802	Нажата	кнопка
		Clear	
cmFileInit	803	Используется объектом	
		TFileDialog	
cmChDir	804	Используется объектом	
		TChDirDialog	
cmRevert	805	Используется объектом	
		TChDirDialog	
cmFileFocused	806	Выбран	новый
		файл	
cmFileDoubleClicked	807	Выбран	новый
		файл	

Модуль VIEWS

В модуле *Views* определено большинство стандартных команд, которые используются как самими отображаемыми объектами, так и разработчиками, например, для проверки статуса операции.

Команда	Значение	Описание
cmValid	0	Проверка созданного объекта
cmQuit	1	Завершение работы
cmError	2	Не используется
cmMenu	3	Выбор элемента меню
cmClose	4	Заккрытие окна
cmZoom	5	Максимизация окна
cmResize	6	Изменение размеров
cmNext	7	Выбор следующего объекта
cmPrev	8	Выбор предыдущего объекта
cmHelp	9	Вызов справочной системы
cmOK	10	Нажата кнопка ОК
cmCancel	11	Нажата кнопка Cancel
cmYes	12	Нажата кнопка Yes
cmNo	13	Нажата кнопка No
cmDefault	14	Нажата кнопка по умолчанию
cmCut	20	\
cmCopy	21	\
cmPaste	22	используются редактором
cmUndo	23	/
cmClear	24	/
cmTile	25	расположение окон
cmCascade	26	расположение окон
cmReceived	50	получение фокуса
Focus		
cmReleased	51	потеря фокуса
Focus		
cmComman dSetChang ed	52	изменение набора команд
cmScrollBar Changed	53	изменения в полосе прокрутки
cmScrollBar Clicked	54	изменения в полосе прокрутки
cmSelectWi ndowNum	55	выбор окна по номеру

Рассмотренные в этой главе объекты играют важную роль в создании интерфейсов прикладных программ, создаваемых с помощью библиотеки Turbo Vision. Окна (объекты на основе *TWindow*) обычно используются для отображения информации, а панели диалога совместно с другими интерфейсными объектами - для ввода данных и уведомления пользователя. Объекты этой группы практически повторяют интерфейсные объекты, существующие в среде Microsoft Windows, что значительно упрощает перенос программ из одной среды в другую. Различные расширения, рассмотренные в этой главе, помогут вам сделать Turbo Vision-программы не только более гибкими, но и практичными.

ГЛАВА 4. Редактор и средства просмотра текста

Два объекта, входящие в комплект Turbo Vision предназначены для работы с текстовыми файлами. Объект *TEditor* реализует сам редактор. Этот редактор позволяет обрабатывать файлы размером до 64 Кбайт, управляется мышью, поддерживает область данными (*clipboard*), командные клавиши, совместимые с редактором WordStar, а также функции поиска и замены. Объект *TTerminal* используется для создания средств для просмотра текста. Ниже рассматриваются основные операции с этими объектами.

Объект TEditor

В приведенном ниже примере показано, что необходимо сделать, чтобы подключить редактор к приложению.

```
////////////////////////////////////////////////////////////////  
EDITOR.PAS - Пример использования объекта TEditor  
////////////////////////////////////////////////////////////////
```

```
{ $M 8192,8192,655360 }
```

```
{ $X +, S- }
```

```
Uses Dos, Objects, Drivers, Memory, Views, Menus,  
Dialogs, StdDlg, MsgBox, App, Editors;
```

```
Const
```

```
HeapSize = 32 * (1024 div 16);
```

```
Type
```

```
PEditorApp = ^TEditorApp;
```

```
TEditorApp = Object(TApplication)
```

```
  constructor Init;
```

```
  procedure HandleEvent(var Event: TEvent); virtual;
```

```
  procedure InitMenuBar; virtual;
```

```
  procedure InitStatusLine; virtual;
```

```
End;
```

```
Function OpenEditor(FileName: FNameStr; Visible: Boolean): PEditWindow;
```

```
var
```

```
  P: PWindow;
```

```
  R: TRect;
```

```
Begin
```

```
  DeskTop^.GetExtent(R);
```

```
  P := New(PEditWindow, Init(R, FileName, wnNoNumber));
```

```
  OpenEditor := PEditWindow(Application^.InsertWindow(P));
```

```
End;
```

```

Constructor TEditorApp.Init;
Begin
  MaxHeapSize := HeapSize;
  inherited Init;
  EditorDialog := StdEditorDialog;
End;

Procedure TEditorApp.HandleEvent(var Event: TEvent);

Procedure FileOpen;
Var
  FileName: FNameStr;
Begin
  FileName := '*.*';
  if ExecuteDialog(New(PFileDialog, Init('*.*', 'Open file',
    '~Name', fdOpenButton, 100)), @FileName) <> cmCancel then
    OpenEditor(FileName, True);
End;

Procedure FileNew;
Begin
  OpenEditor('', True);
End;

Procedure ChangeDir;
Begin
  ExecuteDialog(New(PChDirDialog, Init(cdNormal, 0)), nil);
End;

Begin
  Inherited HandleEvent(Event);
  case Event.What of
    evCommand:
      case Event.Command of
        cmOpen   : FileOpen;
        cmNew    : FileNew;
        cmChangeDir : ChangeDir;
      else
        Exit;
      end;
    else
      Exit;
    end;
  ClearEvent(Event);
End;

Procedure TEditorApp.InitMenuBar;
Var
  R: TRect;
Begin
  GetExtent(R);
  R.B.Y := R.A.Y + 1;
  MenuBar := New(PMenuBar, Init(R, NewMenu(
    NewSubMenu('~F~ile', hcNoContext, NewMenu(
      StdFileMenuItems(
        nil)),
    NewSubMenu('~E~dit', hcNoContext, NewMenu(
      StdEditMenuItems(
        nil)),

```

```

NewSubMenu('W~indows', hcNoContext, NewMenu(
  StdWindowMenuItems(
    Nil)),
Nil)))));
End;

Procedure TEditorApp.InitStatusLine;
Var
  R: TRect;
Begin
  GetExtent(R);
  R.A.Y := R.B.Y - 1;
  New(StatusLine, Init(R,
  - NewStatusDef(0, $FFFF,
    NewStatusKey('~F2~ Save', kbF2, cmSave,
    NewStatusKey('~F3~ Open', kbF3, cmOpen,
    NewStatusKey('~Alt-F3~ Close', kbAltF3, cmClose,
    Nil))),
  Nil)));
End;

Var
  EditorApp: TEditorApp;

Begin
  EditorApp.Init;
  EditorApp.Run;
  EditorApp.Done;
End.

```

Замечание: при выделении памяти для буфера следует помнить:

- параметр *MaxHeapSize* содержит размер буфера в 16-байтовых параграфах. Например, *MaxHeapSize* = 4096 соответствует буферу в 64 Кбайта
- необходимо устанавливать размер буфера в самом начале конструктора объекта-приложения:

```

Constructor TEditorApp.Init;
Begin
  MaxHeapSize := 32 * (1024 div 16);
  inherited Init;
  .....
End;

```

Использование области обмена данными

Область обмена данными используется для обмена фрагментами текста между различными окнами или между различными частями одного окна.

Глобальной переменной *Clipboard* типа *PEditor* необходимо присовить указатель на редактор, как показано ниже:

```
New(ClipWindow, Init(R, "", wnNoNumber));
Clipboard := EditWindow^.Editor;
```

Пример включения поддержки области обмена данными показан ниже. Используется программа *EDITOR.PAS*, изменения отмечены символом "•"

```
{//////////////////////////////////////
EDITOR.PAS - Пример использования объекта TEditor
////////////////////////////////////}
```

```
{ $M 8192,8192,655360 }
{ $X +,S- }
Uses Dos, Objects, Drivers, Memory, Views, Menus, Dialogs,
    StdDlg, MsgBox, App, Editors;

Const
    HeapSize = 32 * (1024 div 16);
{...}
    cmShowClip = 100;
{...}

Type
    PEditorApp = ^TEditorApp;
    TEditorApp = Object(TApplication)
        constructor Init;
        procedure HandleEvent(var Event: TEvent); virtual;
        procedure InitMenuBar; virtual;
        procedure InitStatusLine; virtual;
    End;

{...}
Var
    ClipWindow: PEditWindow;
{...}
Function OpenEditor(FileName: FNameStr; Visible: Boolean): PEditWindow;
var
    P: PWindow;
    R: TRect;
Begin
    DeskTop^.GetExtent(R);
    P := New(PEditWindow, Init(R, FileName, wnNoNumber));
    OpenEditor := PEditWindow(Application^.InsertWindow(P));
End;

Constructor TEditorApp.Init;
Begin
    MaxHeapSize := HeapSize;
    inherited Init;
    EditorDialog := StdEditorDialog;
{...}
    ClipWindow := OpenEditor("", False);
    if ClipWindow <> nil then
```

```

begin
  Clipboard := ClipWindow^.Editor;
  Clipboard^.CanUndo := False;
end;
{...}
End;

Procedure TEditorApp.HandleEvent(var Event: TEvent);

Procedure FileOpen;
Var
  FileName: FNameStr;
Begin
  FileName := '*..*';
  if ExecuteDialog(New(PFileDialog, Init('*..*', 'Open file',
    'N~ame', fdOpenButton, 100)), @FileName) <> cmCancel then
    OpenEditor(FileName, True);
End;

Procedure FileNew;
Begin
  OpenEditor("", True);
End;

Procedure ChangeDir;
Begin
  ExecuteDialog(New(PChDirDialog, Init(cdNormal, 0)), nil);
End;
{...}
Procedure ShowClip;
Begin
  ClipWindow^.Select;
  ClipWindow^.Show;
End;
{...}
Begin
  Inherited HandleEvent(Event);
  case Event.What of
    evCommand:
      case Event.Command of
        cmOpen      : FileOpen;
        cmNew       : FileNew;
        cmChangeDir : ChangeDir;
        cmShowClip  : ShowClip;
      else
        Exit;
      end;
    else
      Exit;
    end;
  ClearEvent(Event);
End;

Procedure TEditorApp.InitMenuBar;
Var
  R: TRect;
Begin
  GetExtent(R);
  R.B.Y := R.A.Y + 1;
  MenuBar := New(PMenuBar, Init(R, NewMenu(

```

```

NewSubMenu('F~ile', hcNoContext, NewMenu(
  StdFileMenuItems(
    nil)),
NewSubMenu('E~dit', hcNoContext, NewMenu(
  StdEditMenuItems(
    nil)),
NewSubMenu('W~indows', hcNoContext, NewMenu(
  StdWindowMenuItems(
    Nil)),
Nil))));
End;

Procedure TEditorApp.InitStatusLine;
Var
  R: TRect;
Begin
  GetExtent(R);
  R.A.Y := R.B.Y - 1;
  New(StatusLine, Init(R,
    NewStatusDef(0, $FFFF,
      NewStatusKey('F2~ Save', kbF2, cmSave,
        NewStatusKey('F3~ Open', kbF3, cmOpen,
          NewStatusKey('Alt-F3~ Close', kbAltF3, cmClose,
            {...}
              NewStatusKey("", kbAltC, cmShowClip,
                {...}
                  Nil)))))
                Nil)));
End;

Var
  EditorApp: TEditorApp;

Begin
  EditorApp.Init;
  EditorApp.Run;
  EditorApp.Done;
End.

```

Еще два объекта, реализованные в модуле *EDITORS*, представляют интерес для разработчиков: это объекты *TMemo* и *TFileEditor*. Объект *TMemo* может использоваться совместно с панелями диалога для реализации мемо-полей.

Объект TMemo

Объект *TMemo* является наследником объекта *TEditor* и предназначен для использования совместно с панелями диалога. Этот объект поддерживает механизм передачи данных с помощью методов *GetData* и *SetData*.

Обычно для установки начальных данных и получения введенных данных, используется запись типа *TMemoData*, которая состоит из двух полей. Поле *Length* содержит

размер данных, которые непосредственно располагаются в буфере, описанном полем *Buffer* типа *TEditBuffer*.

Пример использования объекта *TMemo* приведен ниже.

```
{//////////////////////////////////////
MEMO.PAS - Пример использования объекта TMemo
//////////////////////////////////////}
uses Crt, Objects, App, Drivers, Dialogs, StdDlg, Views, Menus, Editors;
Const
  cmDialog    = 1000;

Type
  TDemoApp = Object(TApplication)
    Dialog  : PDialog;
    procedure InitStatusLine;          virtual;
    procedure HandleEvent(var Event : TEvent);  virtual;
    procedure DemoDialog;
  End;

Procedure TDemoApp.InitStatusLine;
Var
  R : TRect;
Begin
  GetExtent(R);
  R.A.Y := R.B.Y - 1;
  StatusLine := New(PStatusLine, Init(R,
    NewStatusDef(0, $FFFF,
      NewStatusKey('~Alt-X~ Exit', kbAltX, cmQuit,
        NewStatusKey('~Alt-D~ Dialog', kbAltD, cmDialog,
          Nil)),
    Nil)
  ));
End;

Procedure TDemoApp.HandleEvent;
Begin
  Inherited HandleEvent(Event);
  if Event.What = evCommand then
    begin
      case Event.Command of
        cmDialog  : DemoDialog;
      else
        Exit;
      end;
      ClearEvent(Event);
    end;
End;

Procedure TDemoApp.DemoDialog;
Var
  R      : TRect;
  P      : PView;
  hSB    : PScrollBar;
  vSB    : PScrollBar;
  Ind    : PIndicator;
Begin
  R.Assign(10,5,70,15);
```

```

Dialog := New(PDialog, Init(R, 'Memo Dialog'));
With Dialog do
Begin
  R.Assign(13, 8, 38, 9);
  hSB := New(PScrollBar, Init(R));
  Insert(hSB);
  R.Assign(38, 2, 39, 8);
  vSB := New(PScrollBar, Init(R));
  Insert(vSB);
  R.Assign(2, 8, 12, 9);
  Ind := New(PIndicator, Init(R));
  Insert(Ind);
  R.Assign(2, 2, 38, 8);
  P := New(PMemo, Init(R, hSB, vSB, Ind, 2048));
  P.Options := P.Options OR ofFramed;
  Insert(P);
End;
Insert(Dialog);
End;

Var
  DemoApp : TDemoApp;
Begin
  DemoApp.Init;
  DemoApp.Run;
  DemoApp.Done;
End.

```

Отметим, что при инициализации объекта *TMemo* указывается три сопутствующих объекта: две полосы прокрутки (объект *TScrollBar*) - для прокрутки текста по горизонтали и вертикали и объект *TIndicator*, который служит для отображения текущей позиции курсора в файле и режима работы - вставка/замена.

Как и в большинстве случаев использования полос прокрутки с другими объектами, не забудьте поместить сами полосы в панель диалога, используя метод *Insert*.

Объект TFileEditor

Объект *TFileEditor* реализует редактор текстовых файлов. Конструктор этого объекта содержит дополнительный параметр - имя редактируемого файла. Это файл загружается с помощью метода *LoadFile*, который помимо загрузки самого файла устанавливает размер буфера в соответствии с размером файла. Если файла не существует, создается новый файл. Также объект *TFileEditor* содержит методы для сохранения содержимого файла - и сохранения файла с новым именем: *Save* и *SaveAs* соответственно.

Объект *TTerminal* предназначен для создания специальных типов окон, в которые может производиться вывод. При инициализации такого окна помимо стандартных параметров указывается размер буфера. Перед тем как терминальное окно может быть использовано, необходимо ассоциировать текстовый файл с этим окном. Для этого используется процедура *AssignDevice*. В приведенном ниже примере показано, как реализовать терминальное окно, в котором отображаются символы нажатых клавиш. Для расширения функциональности данного примера необходимо изменить метод *TTerminalWindow.HandleEvent*.

```

/////////////////////////////////////////////////////////////////
TERMINAL.PAS: Пример использования объекта TTerminal
/////////////////////////////////////////////////////////////////
uses Objects, Views, App, Drivers, TextView;

Type
PTerminalWindow = ^TTerminalWindow;
TTerminalWindow = Object(TWindow)
  TerminalText : Text;
  Terminal      : PTerminal;
  constructor Init;
  procedure HandleEvent(var Event : TEvent); virtual;
End;

TTerminalApp = Object(TApplication)
  constructor Init;
End;

Constructor TTerminalWindow.Init;
Var
  HScrollBar, VScrollBar : PScrollBar;
  R                      : TRect;
Begin
  Desktop^.GetExtent(R);
  Inherited Init(R, 'Terminal Demo', wnNoNumber);
  R.Grow(-1, -1);
  HScrollBar := StandardScrollBar(sbHorizontal OR sbHandleKeyboard);
  Insert(HScrollBar);
  VScrollBar := StandardScrollBar(sbVertical OR sbHandleKeyboard);
  Insert(VScrollBar);
  New(Terminal, Init(R, HScrollBar, VScrollBar, 1024));
  AssignDevice(TerminalText, Terminal);
  Rewrite(TerminalText);
  Insert(Terminal);
End;

Procedure TTerminalWindow.HandleEvent(var Event : TEvent);
Begin
  If Event.What = evKeyDown then

```

```

Begin
  Write(TerminalText,Event.CharCode);
End
Else Inherited HandleEvent(Event);
End;

Constructor TTerminalApp.Init;
Var
  TextWindow : PTerminalWindow;
Begin
  Inherited Init;
  New(TextWindow,Init);
  InsertWindow(TextWindow);
End;

Var
  TerminalApp : TTerminalApp;
Begin
  TerminalApp.Init;
  TerminalApp.Run;
  TerminalApp.Done;
End.

```

Отображение текста в 16-ричном виде

Комплексный пример, входящий в комплект *Turbo Vision* - *Turbo Vision File Manager*, содержит большое число объектов, которые могут использоваться в ваших приложениях. В модуле *VIEWHEX* содержится объект *THexWindow*, который может быть использован для отображения содержимого любого файла в 16-ричном виде. Пример использования этого объекта приведен ниже.

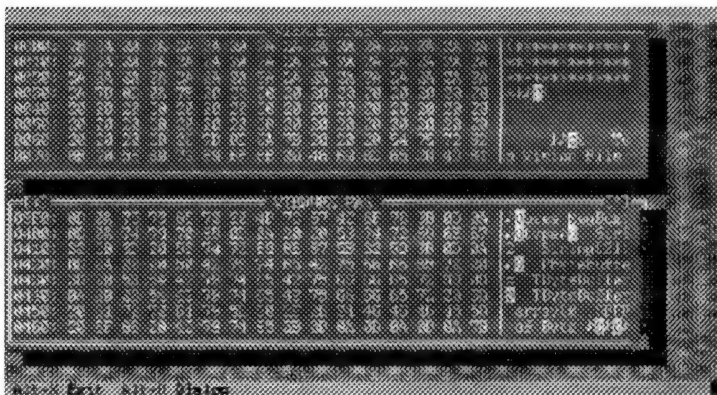


Рис.4.1.Отображение в 16-ричном виде

```

/////////////////////////////////////////////////////////////////
HEXVIEW.PAS - Пример использования объекта THexWindow
/////////////////////////////////////////////////////////////////
uses Crt, Objects, App, Drivers, Dialogs, Views, Menus, ViewHex;
Const
  cmDialog    = 1000;
Type
  TDemoApp = Object(TApplication)
    Dialog  : PDialog;
    procedure InitStatusLine;          virtual;
    procedure HandleEvent(var Event : TEvent);  virtual;
    procedure Demo;
  End;

Procedure TDemoApp.InitStatusLine;
Var
  R : TRect;
Begin
  GetExtent(R);
  R.A.Y := R.B.Y - 1;
  StatusLine := New(PStatusLine, Init(R,
    NewStatusDef(0, $FFFF,
      NewStatusKey('~Alt-X~ Exit', kbAltX, cmQuit,
        NewStatusKey('~Alt-D~ Dialog', kbAltD, cmDialog,
          Nil)),
    Nil)
  ));
End;

Procedure TDemoApp.HandleEvent;
Begin
  Inherited HandleEvent(Event);
  if Event.What = evCommand then
  begin
    case Event.Command of
      cmDialog : Demo;
    else
      Exit;
    end;
    ClearEvent(Event);
  end;
End;

Procedure TDemoApp.Demo;
Var
  R : TRect;
  P : PView;
Begin
  R.Assign(10,5,70,15);
  P := New(PHexWindow, Init(R, 'VIEWHEX.PAS'));
  Insert(P);
End;

Var
  DemoApp : TDemoApp;
Begin
  DemoApp.Init;
  DemoApp.Run;
  DemoApp.Done;
End.

```

Если объединить данный пример со стандартным объектом, позволяющим выбирать имена файлов, то можно создать средство для просмотра содержимого любых файлов.

Для создания средств обычного просмотра ASCII-файлов можно порекомендовать объект *TTextWindow*, содержащийся в модуле *VIEWTEXT*.

Вы можете изучить исходный текст примера *TVFM* чтобы посмотреть, как реализованы эти объекты.

Модуль EDITORS

Модуль *Editors* содержит определения ряда команд, которые используются объектом *TEditor*.

Команда	Значение
cmFind	82
cmReplace	83
cmSearchAgain	84
cmCharLeft	500
cmCharRight	501
cmWordLeft	502
cmWordRight	503
cmLineStart	504
cmLineEnd	505
cmLineUp	506
cmLineDown	507
cmPageUp	508
cmPageDown	509
cmTextStart	510
cmTextEnd	511
cmNewLine	512
cmBackSpace	513
cmDelChar	514
cmDelWord	515
cmDelStart	516
cmDelEnd	517
cmDelLine	518
cmInsMode	519
cmStartSelect	520
cmHideSelect	521
cmIndentMode	522
cmUpdateTitle	523

Практически все команды, определенные в модуле *Editors*, имеют соответствующие методы, которые выполняют некоторые действия. Например, команде *cmNewLine* соответствует метод *NewLine*, команде *cmStartSelect* - метод *StartSelect* и т.д.

ГЛАВА 5. Цветовые палитры

Одной из проблем, которая может возникнуть при использовании объектно-ориентированной библиотеки Turbo Vision, является проблема изменения цветов отображаемых объектов. Приводимые ниже замечания основаны на изучении исходных текстов библиотеки и призваны помочь разработчикам чувствовать себя более уверенно при возникновении потребности в изменении цветов в среде Turbo Vision.

Устройство палитр

Как известно, в Turbo Vision, предком всех отображаемых объектов является объект *TView*. Каждый объект-наследник объекта *TView* имеет метод *Draw*, который служит для отображения этого объекта на экране. Также, каждый отображаемый объект имеет палитру цветов, число элементов в которой зависит от типа объекта: статический текст (объект *TStatic*) имеет палитру из одного элемента, тогда как кнопка (объект *TButton*) - палитру из восьми элементов. Для нахождения палитры отображаемого объекта используется метод *GetPalette* соответствующего объекта.

Рассмотрим устройство метода *Draw* для объекта *TStaticText*. Отметим, что приведенные ниже рассуждения справедливы для всех отображаемых объектов.

Итак, метод *Draw* отображает объект на экране. Прежде всего, внутри этого метода определяется цвет, которым будет отображаться объект. В нашем случае вызывается метод *GetColor(1)*, возвращающий цвет, соответствующий указанному в качестве параметра индексу в палитре. Метод *GetColor* вызывает метод *GetPalette*, возвращающий указатель на палитру объекта *TStaticText*. Из значения, находящегося в палитре, метод *GetColor* определяет, что цвет объекта *TStaticText* на самом деле является 6-м элементом палитры владельца этого объекта и вызывает метод *Owner.GetColor(6)*, который в свою очередь также определяет элемент палитры своего владельца и т.д. до тех пор, пока не будет достигнут "главный" владелец, которым чаще всего бывает объект *TApplication* или его наследник. В объекте *TApplication* и определены реальные цвета, используемые для отображения объектов.

Рассмотрим процесс нахождения цвета более подробно. Как следует из описания объекта *TStaticText*, он имеет палитру из одного элемента, который является шестым элементом в стандартной палитре объекта *TDialog*.

TStaticText -> *TDialog*

Посмотрев описание объекта *TDialog*, мы найдем, что его палитра, состоящая из 32 элементов, занимает с 32 по 63 элемент палитры объекта *TApplication*.

TStaticText -> *TDialog* -> *TApplication*

Таким образом, чтобы изменить цвет отображения объекта *TStaticText*, необходимо изменить значение 6-го элемента палитры объекта *TDesktop*, т.е. 37-го элемента палитры объекта *TApplication*.

Палитра объекта *TApplication* не расписана, но может быть легко получена из файла *APP.PAS*, содержащего интерфейсную часть модуля *APP*, в котором и реализован объект *TApplication*.

В действительности объект *TApplication* содержит определения трех палитр - цветной, черно-белой и монохромной, - используемых в зависимости от режима, установленного при инициализации Turbo Vision). Для простоты мы будем считать, что используется палитра для цветного монитора.

Если мы посмотрим 37-й элемент палитры объекта *TApplication*, то увидим, что его значение равно \$70, что означает черный цвет на сером фоне. Приводимая ниже таблица содержит номера, соответствующие цветам:

0-черный	8-серый
1-синий	9-голубой
2-зеленый	10-светло-зеленый
3-голубой	11-светло-голубой
4-красный	12-светло-красный
5-фиолетовый	13-светло-фиолетовый
6-коричневый	14-желтый
7-белый	15-ярко-белый

Цвета остальных отображаемых объектов могут быть определены описанным выше способом.

Теперь рассмотрим следующую ситуацию: объект типа *TStaticText* помещен не в панель диалога (объект *TDialog*), а в окно (объект *TWindow*). Таким образом, первый (и

единственный элемент палитры объекта *TStaticText*) будет соответствовать 6-му элементу объекта *TWindow* (теперь он является владельцем объекта *TStaticText*). 6-й элемент палитры объекта *TWindow* соответствует 29, 21 или 13 элементу палитры *TApplication*, в зависимости от того какие окна используются (синие, голубые или серые). Если мы посмотрим на значения этих элементов в палитре *TApplication*, то обнаружим, что им соответствуют цвета: желтый на синем (13-й элемент), желтый на голубом (21-й элемент) и черный на сером (29-й элемент). Заметим, что только в случае использования синих окон мы получим тот же цвет объекта *TStaticText*. Таким образом, можно вывести следующее правило

Если отображаемый объект используется не по назначению, это может привести к изменению его цвета.

Те, кто уже пытался работать с палитрами Turbo Vision наверняка сталкивались с ситуацией, когда интерфейсный элемент отображается мигающим белым цветом на красном фоне. Такой эффект происходит в том случае, если в качестве параметра при вызове метода *GetColor* указывается индекс, превышающий размер палитры объекта. В качестве примера рассмотрим, что произойдет в случае, если использовать объект типа *TListBox* внутри окна (объект *TWindow*), а не внутри панели диалога. Отметим, что в руководстве по Turbo Vision указано, что объект *TListBox* использует палитру *TApplication*. Это неверно, используется палитра объекта *TDialog*, которая, как мы увидели выше, "накладывается" на палитру объекта *TApplication*. Объект *TListBox* имеет палитру из пяти элементов, которые соответствуют элементам с 26 по 29 палитры объекта-владельца. В случае, когда *TListBox* используется внутри окна, он будет отображен мигающим белым цветом на красном фоне, что указывает на ошибку. В чем же дело? Рассмотрим палитру объекта *TWindow*. Она состоит из 8 элементов, так что обращение к элементу 26 и выше приводит к выходу за границу палитры *TWindow*.

Изменение цветов

После того как мы разобрались с устройством палитр в Turbo Vision, рассмотрим процесс изменения цветов отображаемых объектов.

Если вы хотите изменить цвет всех объектов данного типа, например, изменить цвет кнопки (объект *TButton*) с зеленого на голубой, вам необходимо внести изменения в

соответствующие элементы палитры объекта *TApplication* (элементы с 41 по 46). Как это делается, показано в приведенном ниже фрагменте.

```
Function TMyApp.GetPalette;
Const
  AppColor : TPalette = CColor;
Begin
  AppColor[41] := # $30;      { Обычный текст }
  AppColor[42] := # $3B;      { По умолчанию }
  AppColor[43] := # $3F;      { Выбранная кнопка }
  AppColor[45] := # $3E;      { Командная клавиша }
  GetPalette := @AppColor;
End;
```

В таблице в конце данной главы приведено назначение всех элементов палитры объекта *TApplication* с указанием цветов по умолчанию.

Более сложной может показаться задача создания нового отображаемого элемента с уникальным набором цветов. Для простоты задачи предположим, что наш новый объект (*TMyView*) использует два цветовых атрибута: один - для отображения обычного текста, а другой - для отображения выделенного текста. Также предположим, что объект *TMyView* будет использоваться внутри объекта *TDialog*. Первое, что нам необходимо сделать в этом случае, - добавить два элемента к палитре объекта *TApplication* (с номерами 64 и 65). Предположим, что первый атрибут должен соответствовать черному цвету на голубом фоне (*\$30*), а второй - белому тексту на голубом фоне (*\$3F*). В таком случае палитра объекта *TApplication* будет выглядеть следующим образом:

```
CColor =
  # $71 # $70 # $78 # $74 # $20 # $28 # $24 # $17 # $1F # $1A # $31
  # $31 # $1E # $71 +
  # $00 # $37 # $3F # $3A # $13 # $13 # $3E # $21 # $00 # $70 # $7F
  # $7A # $13 # $13 +
  # $70 # $7F # $00 # $70 # $7F # $7A # $13 # $13 # $70 # $70 # $7F
  # $7E # $20 # $2B +
  # $2F # $78 # $2E # $70 # $30 # $3F # $3E # $1F # $2F # $1A # $20
  # $72 # $31 # $31 +
  # $30 # $2F # $3E # $31 # $13 # $00 # $00 +
  # $30 # $3F; { <- Два элемента добавлены нами }
```

Отметим, что такие же изменения (с соответствующим атрибутами) должны быть произведены в палитрах *CBlackWhite* и *CMonochrome*.

Затем, мы должны изменить метод *GetPalette* объекта *TDialog*, переопределив его, чтобы он возвращал необходимую палитру:

```
Const
  CNewDialog = CDialog + #64#65;

Type
  TNewDialog = Object (TDialog)
    Function GetPalette: PPalette; virtual;
  End;
.
.
Function TNewDialog.GetPalette: PPalette;

Const
  P: String[Length (CNewDialog)] = CNewDialog;

Begin
  GetPalette := @P;
End;
```

Так как мы добавили два новых цвета в конец стандартной палитры объекта *TDialog*, изначально содержавшей 32 элемента, они станут элементами 33 и 34 палитры объекта *TNewDialog*. Теперь необходимо определить палитру нашего объекта *TMyView*, чтобы он использовал элементы 33 и 34 палитры объекта-владельца.

```
Const
  CMyView = #33#34;

Type
  TMyView = Object (TView)
    Function GetPalette: PPalette; virtual;

  End;

Function TMyView.GetPalette: PPalette;

Const
  P: String[Length (CMyView)] = CMyView;

Begin
  GetPalette := @P;
End;
```

Таким образом, когда метод *Draw* нашего объекта *TMyView* запрашивает атрибут номер 1 (*GetColor(1)*), он получает атрибут с номером 64 и палитры объекта *TApplication*. Для атрибута с номером 2 (*GetColor(2)*) мы

получим атрибут с номером 65 из палитры объекта *TApplication*. Отметим, что если потребуется изменить цвета нашего объекта - нужно просто изменить соответствующие элементы палитры объекта *TApplication*.

Теперь посмотрим, что произойдет, если мы поместим отображаемый объект в другой объект, который не предназначен для этого. Мы уже знаем, что в этом случае произойдет нарушение цветов палитры. Решением этой проблемы в большинстве случаев является создание нового объекта, как показано выше. Таким образом, если мы хотим поместить кнопку (*TButton*) непосредственно в окно (*TWindows*), нам необходимо создать объект-наследник объекта *TButton*, который мы назовем *TWindowButton* и определить для него набор цветов.

В некоторых случаях нет необходимости в расширении палитры объекта *TApplication*. Если нам требуется чтобы кнопка, помещенная в окно выглядела как обычная кнопка, можно использовать уже имеющиеся элементы палитры объекта *TApplication* - с 41 по 46:

```
const
  CNewWindow = CGrayWindow + #41#42#43#44#45#46;
  CWindowButton = #9#10#11#12#13#13#13#14;

type
  TNewWindow = object (TWindow)
    function GetPalette: PPalette; virtual;
    end;

  TWindowButton = object (TButton)
    function GetPalette: PPalette; virtual;
    end;
```

Код метода *GetPalette* аналогичен коду в приведенном выше примере. Теперь, когда метод *TWindowButton.Draw* запрашивает цвет с индексом 2, он накладывается на цвет с индексом 10 объекта *TNewWindow*, который накладывается на цвет с номером 42 в палитре объекта *TApplication*, так же, как если бы кнопка была помещена в объект *TDialog*. Отметим, что в качестве базовой палитры для палитры *CNewWindow* используется палитра *CGrayWindow*, так как объект *TButton* обычно помещается в объект *TDialog*, и два цвета (44 и 46) используют серый фон.

Нам еще осталось рассмотреть следующий вопрос: как быть с объектами, которые используют несколько палитр, как, например, объект *TWindow*: в приложении могут быть синие, серые и голубые окна. Одно из полей объекта *TWindow* - *Palette* используется для указания, какая

цветовая схема используется. Метод *TWindow.GetPalette* выглядит следующим образом:

```
Function TWindow.GetPalette: PPalette;  
  
Const  
  PGray: string[Length (CGrayWindow)] = CGrayWindow;  
  PCyan: string[Length (CCyanWindow)] = CCyanWindow;  
  PBlue: string[Length (CBlueWindow)] = CBlueWindow;  
  
Begin  
  case Palette of  
    wpGrayWindow: GetPalette = @PGray;  
    wpCyanWindow: GetPalette = @PCyan;  
    wpBlueWindow: GetPalette = @PBlue;  
  end;  
End;
```

Такая же техника может быть использована для реализации собственных объектов, которые используют несколько наборов цветов.

Настройка цветов

Помимо статического изменения цветов, реализуемого в момент создания программы, в Turbo Vision возможно динамическое изменение цветов (во время выполнения программы). Для этого используется объект *TColorDialog*. Использование этого объекта показано ниже.

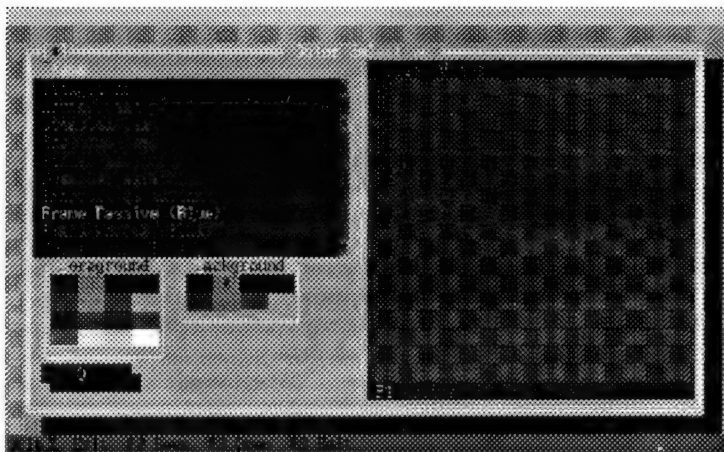


Рис. 5.1. Редатор палитры

```

////////////////////////////////////
Пример использования объекта TColorDialog
Панель диалога изменения цветов содержит два списка,
которые задаются при вызове конструктора TColorDialog.Init.
////////////////////////////////////

```

```
uses App, Objects, Drivers, Menus, Views, ColorSel, Memory;
```

```
Const
```

```
cmColor = 100;
```

```
Type
```

```
TMyApp = Object(TApplication)
```

```
  procedure HandleEvent(var Event : TEvent); virtual;
```

```
  procedure InitStatusLine;
```

```
  procedure Color;
```

```
virtual;
```

```
End;
```

```
Procedure TMyApp.InitStatusLine;
```

```
Var
```

```
R: TRect;
```

```
Begin
```

```
  GetExtent(R);
```

```
  R.A.Y := R.B.Y - 1;
```

```
  New(StatusLine, Init(R,
```

```
    NewStatusDef(0, $EFFF,
```

```
    NewStatusKey('~Alt-X~ Exit ', kbAltX, cmQuit,
```

```
    NewStatusKey('~Alt-C~ Color', kbAltC, cmColor,
```

```
    Nil)),
```

```
    Nil)));
```

```
End;
```

```
Procedure TMyApp.HandleEvent;
```

```
Begin
```

```
  Inherited HandleEvent(Event);
```

```
  if Event.What = evCommand then
```

```
    case Event.Command of
```

```
      cmColor : Color;
```

```
    end;
```

```
End;
```

```
procedure TMyApp.Color;
```

```
var
```

```
ColorDialog : PColorDialog;
```

```
begin
```

```
  ColorDialog := New(PColorDialog, Init(",
```

```
    ColorGroup('Menus',
```

```
      ColorItem('Normal', 2,
```

```
      ColorItem('Disabled', 3,
```

```
      ColorItem('Shortcut', 4,
```

```
      ColorItem('Selected', 5,
```

```
      ColorItem('Selected disabled', 6,
```

```
      ColorItem('Shortcut selected', 7, nil))))),
```

```
    ColorGroup('Dialogs',
```

```
      ColorItem('Frame/background', 33,
```

```
      ColorItem('Frame icons', 34,
```

```
      ColorItem('Scroll bar page', 35,
```

```
      ColorItem('Scroll bar icons', 36,
```

```
      ColorItem('Static text', 37,
```

```
      ColorItem('Button normal', 41,
```

```

ColorItem('Button default', 42,
ColorItem('Button selected', 43,
ColorItem('Button disabled', 44,
ColorItem('Button shortcut', 45,
ColorItem('Button shadow', 46,
Nil ))))));
Nil));

If ValidView(ColorDialog) <> Nil then
Begin
ColorDialog^.SetData(Application^.GetPalette^);
if Desktop^.ExecView(ColorDialog) <> cmCancel then
Begin
Application^.GetPalette^ := ColorDialog^.Pal;
DoneMemory;
ReDraw;
End;
Dispose(ColorDialog, Done);
End;
End;

Var
MyApp: TMyApp;

Begin
MyApp.Init;
MyApp.Run;
MyApp.Done;
End.

```

Примечание: основная палитра, получаемая с помощью метода *Application^.GetPalette^*, заменяется на палитру, формируемую в результате манипуляций с панелью диалога выбора цветов. В нашем случае это - *ColorDialog^.Pal*. После того как палитра изменена, происходит перерисовка всех отображаемых объектов с помощью вызова метода *Draw*. Объект *TColorDialog* содержит связанный список элементов, каждый из которых задается с помощью функции *ColorItem*. При вызове этой функции задается название элемента палитры и индекс цвета этого элемента в глобальной палитре.

Индексы стандартной палитры

#	Элемент	Объект
1	Background	DeskTop
2	Text Normal	Menu
3	Text Disabled	Menu
4	Text Shortcut	Menu
5	Selected Normal	Menu
6	Selected Disabled	Menu

щью
ятру,
алога
осле
всех
Draw.
исок
ощью
ается
нта в

7	Selected Shortcut	Menu
8	Frame Passive	Blue Window
9	Frame Active	Blue Window
10	Frame Icon	Blue Window
11	ScrollBar Page	Blue Window
12	ScrollBar Reserved	Blue Window
13	Scroller Normal Text	Blue Window
14	Scroller Selected Text	Blue Window
15	Reserved	Blue Window
16	Frame Passive	Cyan Window
17	Frame Active	Cyan Window
18	Frame Icon	Cyan Window
19	ScrollBar Page	Cyan Window
20	ScrollBar Reserved	Cyan Window
21	Scroller Normal Text	Cyan Window
22	Scroller Selected Text	Cyan Window
23	Reserved	Cyan Window
24	Frame Passive	Gray Window
25	Frame Active	Gray Window
26	Frame Icon	Gray Window
27	ScrollBar Page	Gray Window
28	ScrollBar Reserved	Gray Window
29	Scroller Normal Text	Gray Window
30	Scroller Selected Text	Gray Window
31	Reserved	Gray Window
32	Frame Passive	Dialog
33	Frame Active	Dialog
34	Frame Icon	Dialog
35	ScrollBar Page	Dialog
36	ScrollBar Controls	Dialog
37	StaticText	Dialog
38	Label Normal	Dialog
39	Label Highlight	Dialog
40	Label Shortcut	Dialog
41	Button Normal	Dialog
42	Button Default	Dialog
43	Button Selected	Dialog
44	Button Disabled	Dialog
45	Button Shortcut	Dialog
46	Button Shadow	Dialog
47	Cluster Normal	Dialog
48	Cluster Selected	Dialog
49	Cluster Shortcut	Dialog
50	InputLine Normal	Dialog
51	InputLine Selected	Dialog
52	InputLine Arrows	Dialog


```

////////////////////////////////////
PALETTE.PAS: Пример динамического изменения палитры
////////////////////////////////////
uses Crt, Objects, App, Drivers, Dialogs, Views, Menus, Colors;
Const
  cmPalette      = 1000;

Type
  TDemoApp = Object(TApplication)
    Dialog : PDialog;
    procedure InitStatusLine; virtual;
    procedure HandleEvent(var Event : TEvent); virtual;
  End;

Procedure TDemoApp.InitStatusLine;
Var
  R : TRect;
Begin
  GetExtent(R);
  R.A.Y := R.B.Y - 1;
  StatusLine := New(PStatusLine, Init(R,
    NewStatusDef(0, $FFFF,
      NewStatusKey('~Alt-X~ Exit', kbAltX, cmQuit,
        NewStatusKey('~Alt-P~ Palette', kbAltP, cmPalette,
          Nil)),
      Nil)
    ));
End;

Procedure TDemoApp.HandleEvent;
Begin
  Inherited HandleEvent(Event);
  if Event.What = evCommand then
  begin
    case Event.Command of
      cmPalette : SelectNewColors;
    else
      Exit;
    end;
    ClearEvent(Event);
  end;
End;

Var
  DemoApp : TDemoApp;
Begin
  DemoApp.Init;
  DemoApp.Run;
  DemoApp.Done;
End.

```

Модуль COLORSEL

В модуле *ColorSel* реализованы объекты, позволяющие динамически изменять цвета интерфейсных элементов.

В этом модуле определен ряд стандартных команд:

Команда	Значение	Описание
cmColorForegroun dChanged	71	Указывает на изменение цвета текста
cmColorBackgroun dChanged	72	Указывает на изменение цвета фона
cmColorSet	73	Указывает на изменение набора цветов
cmNewColorItem	74	Указывает на изменение группы цветов
cmNewColorIndex	75	Указывает на изменение цвета в группе
cmSaveColorIndex	76	Указывает на необходимость сохранения позиции в группе

Эти команды обрабатываются объектами *TColorSelector*, *TMonoSelector*, *TColorDisplay*, *TColorGroupList*, *TColorItemList* и *TColorDialog* и не используются вне этих объектов, за исключением тех случаев, когда создаются объекты-наследники указанных объектов.

Заключение

Использование цветовых палитр является одной из самых "туманных" тем в Turbo Vision. На мой взгляд, разработчики немного перестарались в изобретении универсального способа хранения цветов. В результате, если вы хотите иметь несколько окон разного цвета, вам придется создать объект-наследник для каждого окна, цвет которого должен отличаться от стандартного и переопределить в нем метод *GetPalette*. Оправдано ли это? На мой взгляд, нет. Более того, если новый отображаемый объект использует набор цветов "нестандартного" ряда, как, скажем, объекты, реализующие справочную систему, то приходится расширять основную палитру, как это и сделано в Turbo Vision.

Тем не менее, палитра в Turbo Vision реализована именно так, как я описал в этой главе. Для тех, кто не хочет углубляться в изучение, предлагаю способ динамического изменения цветов отображаемых объектов с помощью объекта *TColorDialog*. В этом случае все проблемы выбора цветов остаются пользователю и не возникает соблазна в использовании таких "сочетаний", как красное с зеленым.

ГЛАВА 6. Модуль VALIDATE.

Объекты проверки ввода

Одной из новинок Turbo Vision 2.0 является реализация ряда объектов проверки ввода (*data validators*). Эти объекты, обычно используемые совместно со строками ввода (объекты типа *TInputLine*), позволяют выполнить проверку корректности вводимых данных на основе задаваемых критериев.

Проверка ввода выполняется в два шага:

- Связывание строки ввода с объектом проверки.
- Вызов метода *Valid*.

Рассмотрим эти шаги более подробно. Каждый объект типа строки ввода имеет поле, которое может содержать указатель на объект проверки ввода. Объекты проверки могут выполнять ряд действий: проверять, попадают ли вводимые данные в указанный диапазон, в список величин или задавать шаблон вводимой информации.

Связывание строки состояния с объектом проверки ввода выполняется следующим образом: сначала создается экземпляр объекта проверки. Так как такие объекты не являются отображаемыми, их конструкторы содержат минимальное число параметров. Например, конструктор объекта проверки вхождения данных в указанный диапазон требует указания всего двух параметров - нижней и верхней границы диапазона:

```
MyRange := New(PRangeValidator, Init(0,99));
```

После того как экземпляр объекта создан, его необходимо связать со строкой ввода. Объекты типа *TInputLine* имеют специальный метод *SetValidator*, который связывает объект проверки ввода с полем *Validator* строки ввода. Обычно создание и присвоение объектов проверки выполняется в одной конструкции:

```
SetValidator(New(PRangeValidator, Init(0,99)));
```

Также можно использовать непосредственный доступ к полю *Validator*:

```
MyInputLine^.Validator := New(PRangeValidator, Init(0,99));
```

После того как объект проверки связан со строкой ввода, проверка вводимых данных происходит автоматически при закрытии окна.

Используя вызов метода *Valid* - метода, который позволяет определить правильность ввода, имеется возможность управления процессом проверки. Метод *Valid* может вызываться в следующих случаях:

- при закрытии окна;
- при потере фокуса;
- при необходимости;

По умолчанию любой отображаемый объект (наследник объекта *TView*) вызывает метод *Valid* при его закрытии. При закрытии панели диалога его метод *Valid* вызывает аналогичные методы всех объектов, расположенных внутри панели диалога. Метод *Valid* панели диалога возвращает значение *True* только в том случае, если все остальные объекты также вернули *True*. Так как строки ввода вызывают метод *Valid* своих объектов проверки, закрытие окна приводит к проверке введенных данных во всех строках ввода.

Проверка вводимых данных при перемещении фокуса выполняется, если у строки ввода установлен флаг *ofValidate*. В этом случае при потере фокуса строка ввода вызывает метод *Valid*, и, если этот метод не возвращает *True*, фокус не перемещается.

Наиболее полезным, на наш взгляд, является проверка при необходимости. В этом случае, проверка выполняется по указанию программиста. Обычно такая проверка выполняется перед сохранением введенных данных:

```
If MyWindow.Valid(cmClose) then  
  MyWindow.GetData(MyData)
```

В приведенном выше примере используется проверка ввода при получении команды *cmClose*. Эти действия можно описать следующим образом: "Я собираюсь закрыть окно, проверь, все ли в порядке".

Ниже, мы рассмотрим объекты проверки ввода, реализованные в Turbo Vision 2.0. Объекты проверки ввода находятся в модуле *VALIDATE.PAS*.

Основой всех объектов проверки ввода является абстрактный объект *TValidator*, задающий основные свойства всех остальных объектов этого типа. На рисунке показана иерархия объектов проверки ввода.

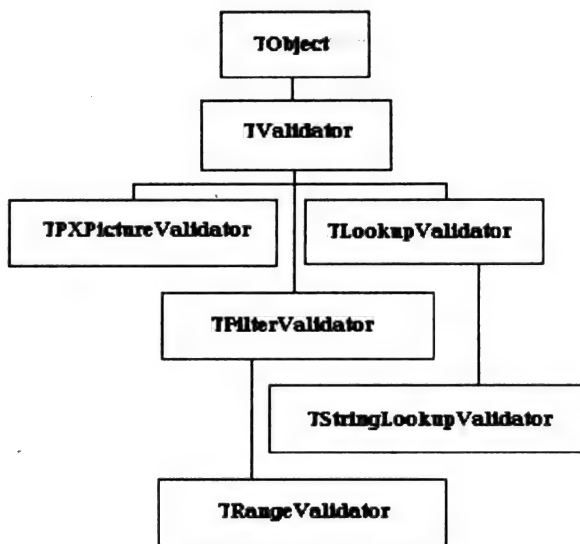


Рис. 6.1. Иерархия объектов проверки ввода

Объект TValidator

TValidator представляет собой абстрактный объект, который служит основой для построения специализированных объектов проверки ввода. Экземпляры объекта *TValidator* не создаются. Ниже мы рассмотрим основные методы этого объекта.

Метод Error

Этот виртуальный метод вызывается методом *Valid* при обнаружении неверно введенной информации. По умолчанию этот метод не выполняет никаких действий, но объекты-наследники переопределяют его для указания пользователю на ошибку.

Метод IsValid

С помощью этого метода проверяется допустимость введенной информации.

Метод *IsValidInput*

Объект строки ввода (типа *TInputLine*) вызывает этот метод после обработки клавиатурных событий. Это дает объекту возможность выполнять проверку по мере ввода (посимвольную проверку). Этот метод чаще всего используется в объектах-фильтрах.

Метод *Valid*

Этот метод возвращает значение *True*, если метод *IsValid* также вернул значение *True*. Иначе вызывается метод *Error*, и *Valid* возвращает *False*. Этот метод вызывается из метода *Valid*, связанного с объектом проверки строки ввода.

Строки ввода, связанные с объектами проверки, вызывают метод *Valid* в двух случаях: если у строки ввода установлен флаг *ofValidate* - в этом случае метод *Valid* вызывается при перемещении фокуса, или при закрытии панели диалога, в этом случае вызываются методы *Valid* всех дочерних элементов управления.

Объект *TFilterValidator*

Этот объект реализует фильтр вводимой информации. При создании такого объекта указывается набор допустимых символов. Такой набор располагается в поле *ValidChars*, имеющем тип *TCharSet*. Например, для разрешения ввода только цифр поле *ValidChars* должно содержать значение ['0'..'9'].

Метод *IsValidInput*

Проверяет каждый символ строки на принадлежность набору *ValidChar*.

Объект *TPXPictureValidator*

С помощью этого объекта реализуется проверка вводимых данных по указанному шаблону. Используются те же форматы описания шаблонов, что и в СУБД Paradox фирмы Borland.

Поле *Pic* типа *PString* содержит спецификацию шаблона. Значение этого поля задается при вызове конструктора объекта.

Метод IsValidInput

Выполняет сравнение введенной строки с шаблоном. Ряд символов в строке может быть изменен, если значение параметра *NoAutoFill* равно *True*.

Метод Picture

Выполняет форматирование строки в соответствии с указанным шаблоном. Основные символы, используемые при задании шаблонов, приведены в таблице. Более подробно создание и использование шаблонов описано в документации по СУБД Paradox.

Символ	Назначение
#	Допустимы только цифры
?	Допустимы только буквы
&	Допустимы только буквы, ввод преобразуется к символам верхнего регистра
@	Допустимы любые символы
!	Допустимы любые символы, ввод преобразуется к символам верхнего регистра
:	Следующий символ используется буквально
.	Число повторений
[]	Опциональные символы
{}	Групповый символы
,	Альтернативные символы

Все остальные символы используются буквально.

Ниже приводится ряд примеров использования шаблонов:

```
StockNum := New(PPxPictureValidator, Init('&&&-# # #', True));
```

Задается последовательность из трех букв и трех цифр, разделитель "-" вводится автоматически.

```
DateField := New(PPxPictureValidator, Init('{ # | # | } / { # | # | } / { # # | # # | }', True);
```

Задается ввод даты в формате ММ/ДД/ГГ, разделители "/" вводятся автоматически.

Объект *TRangeValidator*

Этот объект выполняет проверку на попадание вводимых данных в диапазон, границы которого задаются полями *Min* и *Max*.

Объект *TLookupValidator*

Объект этого типа производит сравнение вводимой пользователем строки со списком допустимых значений. Этот объект является абстрактным. Его экземпляры не создаются. На его основе построен объект *TStringLookupValidator*. При создании объекта на базе *TLookupValidator*, указывается список допустимых значений и переопределяется метод *Lookup*, который должен возвращать значение *True* только в том случае, если введенные пользователем данные соответствуют данным списка.

Объект *TStringLookupValidator*

Этот объект, построенный на основе объекта *TLookupValidator*, выполняет поиск введенной пользователем строки в списке допустимых строк (хранимом как коллекция). Объекты такого типа используются для ввода строки, являющейся подмножеством набора строк.

В завершении рассмотрения объектов модуля *VALIDATE* приведем пример их использования в прикладной программе. В демонстрационной программе создается панель диалога, содержащая три поля (строки ввода). В первом поле возможен ввод только цифр от 100 до 999, во втором - только символов верхнего регистра, а в третьем - номера телефона. Переопределенные методы *Error* соответствующих объектов позволяют выводить сообщения об ошибках на русском языке.

```
{-----  
Пример использования объектов проверки данных  
-----}
```

```
uses Objects, App, Dialogs, Views, Menus, Drivers, MsgBox, Validate;
```

```
Const  
  cmNewDlg = 100;
```

```

Type
TMyApp = Object(TApplication)
  procedure InitStatusLine; virtual;
  procedure HandleEvent(var Event : TEvent); virtual;
  procedure NewDlg;
End;

PMyRangeValidator = ^TMyRangeValidator;
TMyRangeValidator = Object(TRangeValidator)
{
  Переопределение процедуры для вывода
  сообщения об ошибке
}
  procedure Error; virtual;
End;

PMyPXPictureValidator = ^TMyPXPictureValidator;
TMyPXPictureValidator = Object(TPXPictureValidator)
{
  Переопределение процедуры для вывода
  сообщения об ошибке
}
  procedure Error; virtual;
End;

Procedure TMyRangeValidator.Error;
Var
  Params: Array[0..1] of LongInt;
Begin
  Params[0] := Min; Params[1] := Max;
  MessageBox( #3'Значение вне диапазона %d - %d', @Params,
    mfError + mfOKButton);
End;

Procedure TMyPXPictureValidator.Error;
Begin
  MessageBox( #3'Ошибка в формате данных. Шаблон: '#13#3' %s', @Pic,
    mfError + mfOKButton);
End;

Procedure TMyApp.InitStatusLine;
Var
  R : TRect;
Begin
  GetExtent(R);
  R.A.Y := R.B.Y - 1;
  StatusLine := New(PStatusLine, Init(R,
    NewStatusDef(0,$FFFF,
    NewStatusKey('~Alt-X~ Exit', kbAltX, cmQuit,
    NewStatusKey('~Alt-D~ Dialog', kbAltD, cmNewDlg,
    nil)),
    nil)
  ));
End;

Procedure TMyApp.NewDlg;
Var
  Dialog : PDialog;
  R : TRect;

```

Input : PInputLine;

Control : Word;

Begin

R.Assign(20,5,60,20);

Dialog := New(PDialog,Init(R,'Data Validation'));

With Dialog do

Begin

R.Assign(33,3,38,4);

Input := New(PInputLine,Init(R,3));

{-----}

Возможен ввод только цифр в диапазоне от 100 до 999

{-----}

Input.Validator := New(PMyRangeValidator, Init(100, 999));

Insert(Input);

R.Assign(8,3,31,4);

Insert(New(PLabel,Init(R,'Code: 100 to 999',Input)));

R.Assign(2,6,38,7);

Input := New(PInputLine,Init(R,30));

{-----}

Возможен ввод только символов верхнего регистра

{-----}

Input.Validator := New(PFilterValidator,Init(['A'..'Z','A'..'Я']));

Insert(Input);

R.Assign(2,5,37,6);

Insert(New(PLabel,Init(R,'Only uppercase characters allowed',Input)));

R.Assign(26,8,38,9);

Input := New(PInputLine,Init(R,9));

{-----}

Возможен ввод по указанному шаблону. Последний параметр конструктора управляет режимом атоввода разделителей.

{-----}

Input.Validator := New(PMyPXPictureValidator,Init('###-##-##',True));

Insert(Input);

R.Assign(2,8,24,9);

Insert(New(PLabel,Init(R,'Phone : ###-##-##',Input)));

R.Assign(10,11,20,13);

Insert(New(PButton,Init(R,'Ок',cmOk,bfDefault)));

R.Assign(20,11,30,13);

Insert(New(PButton,Init(R,'Cancel',cmCancel,bfNormal)));

{-----}

Установить фокус на первую строку ввода

{-----}

SelectNext(False);

End;

Control := DeskTop.ExecView(Dialog);

{-----}

Проверка введенных данных выполняется по нажатию кнопки ОК. В случае ошибки, панель диалога остается открытой, а фокус будет установлен на строку с ошибочным значением

{-----}

Valid(cmOk);

End;

```

Procedure TMyApp.HandleEvent;
{-----
  Обрабатывается только одна команда - cmNewDlg
-----}
Begin
  inherited HandleEvent(Event);
  if Event.What = evCommand then if Event.Command = cmNewDlg then
    NewDlg else Exit;
  ClearEvent(Event);
End;

Var
  MyApp : TMyApp;

Begin
  MyApp.Init;
  MyApp.Run;
  MyApp.Done;
End.

```

Заключение

Введение объектов проверки ввода существенно облегчило контроль за действиями пользователя и позволило решать проблемы фильтрации вводимой информации более простыми способами. Объекты, включенные в модуль *VALIDATE*, могут использоваться как в программах, создаваемых на базе Turbo Vision, так и Windows-программах, что позволяет походить к пользовательским интерфейсам с одинаковой точки зрения. Для тех, кто по каким-либо причинам не желает использовать объекты проверки ввода, можно порекомендовать различные расширения и изменения свойств объекта *TInputLine*, приведенные в главе 3.

ГЛАВА 7. Использование справочной системы

Наличие контекстно-зависимой справочной системы делает прикладную программу более удобной для пользователя и придает ей профессиональный вид. Библиотека Turbo Vision включает в себя все необходимое для создания контекстно-зависимой справочной системы. Создание и использование простейшей справочной системы мы и рассмотрим ниже.

Каждый отображаемый объект Turbo Vision (наследник объекта *TView*) содержит поле *HelpCtx*, которое может содержать индекс экрана справочной системы. По умолчанию значение этого поля равно *hcNoContext*, значит, для этого объекта нет справочного экрана. Используя метод *GetHelpCtx*, для каждого отображаемого объекта можно определить индекс справочного экрана, который передается конструктору справочного окна. Вся работа со справочными окнами определена в модуле *HelpFile*, входящем в комплект поставки Turbo Vision. Таким образом, для подключения справочной системы к прикладной программе необходимо выполнить следующие шаги:

- Присвоить каждому отображаемому объекту уникальное значение *HelpCtx*, которое будет использоваться для вызова справочного экрана с описанием данного объекта.
- Создать метод (назовем его *Help*), инициализирующий и отображающий на экране справочное окно с текстом для указанного контекста.
- Создать текст справочной системы.
- Преобразовать текстовый файл в специальный формат при помощи утилиты *TVHC.EXE*

Сначала приведем небольшую программу, использующую справочную систему, а затем дадим необходимые комментарии.

```
{-----  
HELPDEMO.PAS - Пример использования справочной системы  
-----}
```

```
uses
```

```
App
```

```
{объект TApplication}
```

```

Objects,      {TRect}
Drivers,      {события и их обработка}
Dialogs,      {панели диалога}
Menus,        {меню и строка состояния}
Views,        {отображаемые объекты}
HelpFile,     {справочная система}
Help;         {индексы справочной системы}

Const
cmOk      = 1;
cmCancel  = 2;
cmHelp    = 3;      {Вызов справочной системы}
IsHelp    : Boolean = False;  {Справочное окно открыто ?}

```

```

Type
TMyApp = Object(TApplication)
Dialog : PDialog;
constructor Init;
procedure InitStatusLine; virtual;
function GetPalette: PPalette; virtual;
procedure Help;
procedure HandleEvent(var Event: TEvent); virtual;
End;

```

Constructor TMyApp.Init;

```

Var
R      : TRect;
OkButton : PButton;
CancelButton : PButton;

```

Begin

```

TApplication.Init;
RegisterHelpFile;
R.Assign(20,5,60,20);
Dialog := New(PDialog,Init(R,'Dialog'));
With Dialog^ do
Begin
R.Assign(10,12,20,14);
OkButton := New(PButton,Init(R,'ОК',cmOk,bfDefault));
{Установить значение справочного контекста}
OkButton.HelpCtx := hccmOk;
Insert(OkButton);
R.Assign(20,12,30,14);
CancelButton := New(PButton,Init(R,'Сancel',cmCancel,bfNormal));
{Установить значение справочного контекста}
CancelButton.HelpCtx := hccmCancel;
Insert(CancelButton);
End;
Insert(Dialog);
End;

```

Procedure TMyApp.InitStatusLine;

```

Var
R : TRect;
Begin
GetExtent(R);
R.A.Y := R.B.Y - 1;
StatusLine := New(PStatusLine, Init(R,
NewStatusDef(0,$FFFF,

```

```
NewStatusKey('~Alt-X~ Exit',kbAltX,cmQuit,
```

{Нажатие клавиши F1 вызывает контекстно-зависимую справочную систему}

```
NewStatusKey('~F1~ Help',kbF1,cmHelp,
nil)),
nil)
));
End;
{
```

Это необходимо для нормального отображения справочного окна. Значение стандартной палитры цветов расширяется палитрой справочных окон

```
}
Function TMyApp.GetPalette;
Const MyColor: TPalette = CAppColor + CHelpColor;
Begin
  GetPalette := @MyColor;
End;

Procedure TMyApp.Help;
Var
  HWnd    : PWindow;
  HFile   : PHelpFile;
  HStream : PDosStream;

Begin
  If Not IsHelp Then
    Begin
      IsHelp := True;
      HStream := New(PDosStream, Init('HELP.HLP', stOpenRead));
      HFile := New(PHelpFile, Init(HStream));
      If HStream^.Status <> stOk Then
        Begin
          Dispose(HFile, Done);
        End
      Else
        Begin
          {Определить текущий контекст}
          HWnd := New(PHelpWindow, Init(HFile, GetHelpCtx));
          ExecView(HWnd);
          Dispose(HWnd, Done);
        End;
      IsHelp := False;
    End;
End;

Procedure TMyApp.HandleEvent;
Begin
  TApplication.HandleEvent(Event);
  if Event.What = evCommand then
    Begin
      case Event.Command of
        cmHelp : Help;
      end;
    End;
```

```

End;
ClearEvent(Event);
End;

Var
MyApp : TMyApp;

Begin
MyApp.Init;
MyApp.Run;
MyApp.Done;

End.

```

В приведенном выше примере создается панель диалога, содержащая две кнопки - Ok и Cancel. Кнопкам присваивается справочный контекст *hccmOk* и *hccmCancel* соответственно. Во всех остальных случаях используется контекст по умолчанию (значение *hcNoContext*). Для работы со справочной системой мы используем метод *HandleEvent*. В этом методе после получения команды *cmHelp* (посылаемой клавишей F1, определенной в методе *InitStatusLine*) мы вызываем метод *Help*. В этом методе мы проверяем, является ли справочная система активной (значение переменной *IsHelp*) и если справочный экран не отображен на экране, инициализируем поток (файл *HELP.HLP*) и отображаем окно типа *THelpWindow*. Окно такого типа "знает" реакцию на все стандартные действия пользователя.

Справочный файл создается как обычный *ASCII*-файл (для его создания может быть использован редактор среды разработчика), а затем компилируется с помощью компилятора *TVHC*, входящего в комплект поставки Turbo Vision. Текст, используемый в нашем примере, приводится ниже

;Пример справочной системы для Turbo Vision

```

.topic NoContext = 0

```

К сожалению, для этого контекста нет дополнительной информации

```

.topic cmOk = 1

```

Нажатие кнопки ОК приводит к отправке команды *cmOk*

```

.topic cmCancel = 2

```

Нажатие кнопки Cancel приводит к послылке команды *cmCancel*:

Как видно из приведенного примера, каждому экрану (раздел *.topic*) присваивается уникальный номер. Внутри текста может быть ссылка на другой экран, которая указывается следующим образом:

{название ссылки : название экрана}

Ссылка отображается соответствующим атрибутом, и при ее активации происходит переход на указанный экран.

Отметим, что компилятор *TVHC* создает файл *xxxx.PAS* (где *xxxx* - имя файла, содержащего текст справочной системы), в котором указываются все константы справочных экранов. Этот файл может быть использован в прикладной программе для отождествления справочных контекстов отображаемых объектов с экранами справочной системы.

Теперь покажем способ использования "двуязычной" справочной системы. Непростая на первый взгляд задача решается тривиальным способом: при создании справочного текста, мы используем номера экранов, скажем в диапазоне от 0 до 199 для англоязычной справки, а номера в диапазоне от 200 до 399 - для русскоязычной (т.е. те же номера, увеличенные на 200). Глобальная переменная хранит текущее состояние - в каком режиме мы находимся. Все, что необходимо сделать при вызове справочной системы - это передать индекс *GetHelpCtx* в одном случае или *GetHelpCtx* + 200 в другом. Поясним это утверждение фрагментом кода.

Метод HELP - пример "двуязычной" справочной системы
-----}

```
Procedure TMyApp.Help;
Var
  HWnd   : PWindow;
  HFile  : PHelpFile;
  HStream : PDocStream;

Begin
  If Not IsHelp Then
  Begin
    IsHelp := True;
    HStream := New(PDocStream, Init('HELP.HLP', stOpenRead));
    HFile := New(PHelpFile, Init(HStream));
    If HStream^.Status <> stOk Then
```

```

Begin
  Dispose(HFile, Done);
End
Else
Begin
  {Определить текущее состояние-переменная Lat}
  If Lat then
    {Вызвать справку на английском языке}
    HWnd := New(PHelpWindow, Init(HFile, GetHelpCtx))
  else
    {Вызвать справку на русском языке}
    HWnd := New(PHelpWindow, Init(HFile, GetHelpCtx + 200))
  ExecView(HWnd);
  Dispose(HWnd, Done);
End;
IsHelp := False;
End;
End;

```

Примечание: переменная *Lat* (тип *Boolean*) содержит текущее состояние системы (используется ли английский или русский текст в меню и строке состояния). В зависимости от ее состояния используется либо индекс, возвращаемый функцией *GetHelpCtx*, либо индекс, увеличенный на значение диапазона (в данном примере - 200).

Некоторые замечания по расширению справочной системы

Ниже приводится описание способа расширения справочной системы для Turbo Vision в результате которого при нажатии клавиши Alt-F1 станет возможным возврат. Посмотрим, как реализовать поддержку до 8 экранов.

1. В описании констант сразу же после директивы *uses* вставить строку:

```
MaxStack = 8; { максимальное число сохраняемых экранов }
```

2. В описании объекта *THelpViewer* необходимо поместить строку:

```
Stack : array[1..MaxStack] of Integer;
```

3. В конструкторе *THelpViewer.Init* завести переменную

```
I : Integer;
```

Добавить следующий текст в конце реализации конструктора:

```
For I := 1 to MaxStack do Stack[I] := 0;  
Stack[1] := Context;
```

4. В теле процедуры *THelpViewer.HandleEvent* добавить две новых процедуры:

{Процедура Push - помещение в стек}

```
Procedure Push(Element : Integer);  
Var I : Integer;  
Begin  
  For I := MaxStack DownTo 2 Do Stack[I] := Stack[I-1];  
  Stack[1] := Element;  
End;
```

{Функция Pop - извлечение из стека}

```
Function Pop : Integer;  
Var I : Integer;  
Begin  
  For I := 1 to (MaxStack-1) do Stack[I] := Stack[I + 1];  
  Stack[MaxStack] := 0;  
  Pop := Stack[1];  
End;
```

5. В обработчике событий добавить обработку следующего события:

```
kbAltF1: SwitchToTopic(Pop);
```

Заключение

Наличие справочной системы придает приложению не только профессиональный вид - оно существенно облегчает жизнь пользователя. Средства, предоставляемые Turbo Vision, обеспечивают простой в использовании и эффективный механизм создания справочных систем. Для любителей экспериментов могу порекомендовать следующий способ расширения объектов, включенных в модуль *HelpFile*: разработайте наследники этих объектов, которые бы работали с упакованными справочными файлами. Программный продукт "*Open Architecture*", распространяемый фирмой Borland, содержит описание формата справочных файлов, которые могут подключаться

к интегрированной среде разработчика, а также компилятор для создания таких файлов.

Еще одно замечание: разрабатывайте справочную систему по мере создания самого приложения, параллельно с написанием документации.

ГЛАВА 8. Отладка Turbo Vision-приложений

В этом разделе мы рассмотрим некоторые вопросы отладки приложений, создаваемых при помощи объектно-ориентированной библиотеки Turbo Vision.

Отладка приложений, использующих несколько отличную от стандартной модель обработки пользовательского ввода, может потребовать специального подхода. Характер отладки может зависеть от типа возникающих ошибок. Чаще всего ошибки в Turbo Vision-программах возникают из-за неправильной обработки потока входных сообщений. В этом случае можно порекомендовать использование функции *MessageBox* в методах *HandleEvent* тех объектов, работа которых вызывает сомнения. Например:

///
WINDOW: Пример использования объекта TWindow
Отладочная версия

///}

```
uses App, StdDlg, Objects, Drivers, Menus, Views, Dialogs, MsgBox;  
{ $DEFINE DEBUG }
```

```
Const
```

```
  cmNewWindow = 3000;
```

```
Type
```

```
  TDemoApp = Object(TApplication)
```

```
    procedure InitStatusLine;          virtual;
```

```
    procedure HandleEvent(var Event : TEvent);  virtual;
```

```
    procedure NewWindow;
```

```
  End;
```

```
Var
```

```
  W : PWindow;
```

```
Procedure TDemoApp.InitStatusLine;
```

```
Var
```

```
  R : TRect;
```

```
Begin
```

```
  GetExtent(R);
```

```
  R.A.Y := R.B.Y-1;
```

```
  StatusLine := New(PStatusLine, Init(R,
```

```
    NewStatusDef(0, $FFFF,
```

```
    NewStatusKey('~Alt-X~ Exit', kbAltX, cmQuit,
```

```
    NewStatusKey('~Alt-W~      kbAltW, cmNewWindow,
```

```
    nil)),
```

```
    nil)
```

```
  ));
```

```
End;
```

```
Procedure TDemoApp.HandleEvent;
```

```
Begin
```

```
  Inherited HandleEvent(Event);
```

```

If Event.What = evCommand Then
Begin
  If Event.Command = cmNewWindow Then
  Begin
    {$IFDEF DEBUG}
    MessageBox('C:\cmNewWindow', Nil, mfOkButton OR mfInformation);
    {$ENDIF}
    NewWindow;
    End;
    ClearEvent(Event);
  End
End;
Procedure TDemoApp.NewWindow;
Var
  R : TRect;
Begin
  R.Assign(20,5,60,15);
  W := New(PWindow, Init(R, ", wnNoNumber));
  InsertWindow(W);
End;

Var
  DemoApp : TDemoApp;
Begin
  DemoApp.Init;
  DemoApp.Run;
  DemoApp.Done;
End.

```

Другой, не менее распространенной ошибкой является неверное управление памятью: всякий раз, когда объект больше не требуется, его необходимо удалить из памяти, используя деструктор *Done*. Парное использование конструктора *Init* и деструктора *Done* является хорошим стилем программирования независимо от того, что в ряде случаев вызов деструкторов необязателен.

В примерах, поставляемых вместе с *Turbo Vision*, есть модуль *Gadgets*, в котором реализован объект *THeapView*, использование которого может помочь при отладке программ. Этот объект реализует средство для отображения текущего размера кучи. Пример использования этого объекта показан ниже.

```

{////////////////////////////////////}
Пример использования объекта THeapView;
{////////////////////////////////////}
Uses
  Objects, Drivers, Views, Menus, Dialogs, App, Gadgets;

{$DEFINE DEBUG}

Type
  PDemo = ^TDemo;
  TDemo = Object(TApplication)
  {$IFDEF DEBUG}

```

```

    Heap: PHeapView;
    Procedure Idle;                                virtual;
{$ENDIF}
    Constructor Init;
    Procedure InitStatusLine;                    virtual;
End;

Constructor TDemo.Init;
Var
    R: TRect;
Begin
    Inherited Init;
{$IFDEF DEBUG}
    GetExtent(R);
    Dec(R.B.X);
    R.A.X := R.B.X - 9; R.A.Y := R.B.Y - 1;
    Heap := New(PHeapView, Init(R));
    Insert(Heap);
{$ENDIF}
End;

{$IFDEF DEBUG}
Procedure TDemo.Idle;
Begin
    Inherited Idle;
    Heap^.Update;
End;
{$ENDIF}

Procedure TDemo.InitStatusLine;
Var
    R: TRect;
Begin
    GetExtent(R);
    R.A.Y := R.B.Y - 1;
    StatusLine := New(PStatusLine, Init(R,
        NewStatusDef(0, $FFFF,
            NewStatusKey('~Alt-X~ Exit', kbAltX, cmQuit,
                Nil),
                Nil)));
End;

Var
    Demo: TDemo;

Begin
    Demo.Init;
    Demo.Run;
    Demo.Done;
End.

```

Обратите внимание на использование директив *\$DEFINE*, *\$IFDEF* и *\$ENDIF*. С их помощью можно одновременно создавать и отладочную, и финальную версии программы. Так, в приведенном выше примере, удалив директиву *\$DEFINE DEBUG*, мы получим финальную версию программы. Отмечу, что использование

объекта *THearView* в рабочей версии программы не является необходимым и будет только отвлекать пользователя.

Если же в процессе создания и отладки программы у вас все же возникают проблемы, вам смогут помочь некоторые из рассмотренных ниже модулей, специально созданных для облегчения процесса отладки Turbo Vision-приложений.

Модуль TVDEBUG

В состав Turbo Vision 2.0 входит специальное средство - *TVDEBUG*. Этот модуль содержит ряд объектов, которые позволяют выполнять отладку приложений. Объект *TApplication* является наследником объекта *TApplication*, реализованного в модуле *APP*. В нем переопределяется метод *GetEvent*, который возвращает следующее имеющееся в системе событие.

В приведенном ниже примере показана минимальная программа, при запуске которой отображаются два окна: окно, в котором показываются все события, происходящие в системе, и окно, в котором могут появляться текстовые сообщения, посылаемые при помощи функций *Write* и *Writeln*.

```
uses Drivers, Objects, Views, Menus, App, TVDebug;  
Var  
  DebugApp : TApplication;  
Begin  
  DebugApp.Init;  
  DebugApp.Run;  
  DebugApp.Done;  
End.
```

В модуле *TVDEBUG* реализованы два объекта - *TEventWindow* и *TLogWindow* - наследники объектов *TTextWindow* и *TWindows* соответственно, которые предназначены для отображения различной отладочной информации. Эти два окна создаются автоматически при инициализации отладочной версии объекта *TApplication*.

Окно *TEventWindow* отображает все события, происходящие в системе. Меню *Options* позволяет установить какие события будут отображаться в этом окне. По умолчанию отображаются все события:

```
Filters := evMouse or evKeyBoard or evMessage;
```

В ряде случаев имеет смысл отключить отображение событий от мыши (особенно события, связанного с

перемещением мыши), так как они сильно перегружают содержимое протокола. Если модуль *TVDEBUG* подключается после модуля *VIEWS*, то также отображаются все вызовы функции *Message* (она переопределена). Необходимо отметить, что допустимо использование только одного окна данного типа. Панель диалога, отображаемая при вызове команды *Options/Filters*, позволяет включить или отключить регистрацию следующих сообщений:

- нажатие кнопки мыши (*Mouse Down*);
- отжатие кнопки мыши (*Mouse Up*);
- перемещение мыши (*Mouse Move*);
- повторяющиеся манипуляции с мышью (*Mouse Auto*);
- нажатие клавиши (*Keyboard*);
- командное событие (*Command*);
- переданное событие (*Broadcast*).

Окно *TLogWindow* предназначено для отображения всех текстовых сообщений, посылаемых при помощи функций *Write* и *Writeln* в окне. Так же, как и в случае с окном *TEventWindow*, возможно создание только одного окна данного типа.

Модуль *TVDEBUG* не является идеальным средством для отладки приложений, созданных с помощью Turbo Vision, но его использование в ряде случаев может помочь быстрее локализовать ошибку, чем при использовании отладчика. Отмечу, что преимуществом данного подхода является то, что модуль *TVDEBUG*, а также сопутствующие модули - *CMDNAMER* и *KEYNAMER*, поставляются в исходном виде, что делает возможным их настройку на более конкретные задачи.

Средства, входящие в Turbo Vision Development Kit

Еще одним средством для отладки приложений, созданных с помощью Turbo Vision является набор модулей, входящих в состав программного продукта Turbo Vision Development Kit, разработанного фирмой Blaise Computing Inc.

В этом пакете также поставляется отладочная версия объекта *TApplication*, но с более широкими возможностями. Отладочная версия объекта *TApplication*, называемая

bTApplication, предназначена для замены обычного объекта *TApplication* во время отладки программ. Даже такая минимальная программа, как та, что показана ниже, позволяет посмотреть, что предоставляет этот отладочный объект.

```
//////////////////////////////////////////////////////////////////
Подключение отладочной версии объекта
TApplication фирмы Blaise Computing Inc.
//////////////////////////////////////////////////////////////////
```

```
Uses betaBApp;
Var
  MyApp : bTApplication;
Begin
  MyApp.Init;
  MyApp.Run;
  MyApp.Done;
End.
```

При включении объекта *bTApplication* вместо объекта *TApplication* в полосе меню появляется еще один элемент, который открывает доступ к отладочному меню. Это меню предоставляет следующие возможности для отладки:

- запись/воспроизведение событий (подменю *Record/playback*;
- просмотр событий (подменю *Event viewer*);
- просмотр памяти (подменю *Memory viewer*);
- просмотр размера кучи (подменю *maxAvail viewer*).

Окно просмотра событий также имеет панель настройки, в которой можно указать, какие события должны отображаться. Помимо стандартных событий также могут отображаться пользовательские события, коды которых попадают в один из диапазонов, указанных в панели настройки.

Окно просмотра данных о различных областях памяти программы выглядит следующим образом:

[.]----- Memory Usage Information -----			
--- Segment Information ---			
PSP	: \$09E6	Code Size	: 99136 (\$18340) bytes
Data Segment	: \$222A	Data Size	: 7776 (\$1E60) bytes
Stack Segment	: \$2418	Stack Size	: 16384 (\$4000) bytes
Stack Pointer	: \$3F16	Heap Size	: 491248 (\$77EF0) bytes
--- Heap Information ---			
Heap Origin	: 2810:0000	Available	: 481224 (\$757C8) bytes
Heap End	: 9FFF:0000	Allocated	: 10024 (\$82728) bytes
Heap Top	: 2A82:0000	% Used	: 2%
First Free	: 2A82:0000	In Blocks	: 0 (\$00000) bytes
High Heap	: 2A82:0000	Blocks	: 0
--- FreeList Information ---			
		Address	Size
Copyright (c) 1991			
Blaise Computing Inc.			

При выборе команды *maxAvail viewer* имеется возможность отображения значения переменной *MaxAvail* в статусной строке как в десятичном, так и в шестнадцатиричном виде, что задается специальной опцией подменю.

По сравнению со средствами модуля *TVDEBUG* новой здесь является возможность записи/воспроизведения событий и окно для просмотра памяти.

ГЛАВА 9. Неотображаемые объекты

Неотображаемые объекты располагаются в отдельной ветви иерархии объектов Turbo Vision. Предком этих объектов, как и всех объектов Turbo Vision, является абстрактный объект *TObject*. К неотображаемым объектам относятся коллекции, потоки и ресурсы. Неотображаемые объекты могут использоваться в программах, которые не используют отображаемых объектов Turbo Vision. Так, потоки удобно использовать при работе с файлами, коллекции - для хранения данных, списки строк - для хранения наборов строк, доступных по "ключу", и т.д.

Коллекции

Коллекция - это структура данных, в которой могут храниться другие данные. В отличие от традиционных структур типа массивов, коллекции обладают рядом существенных преимуществ:

- размер коллекций может динамически изменяться;
- коллекции могут содержать элементы различных типов;
- помимо объектов коллекции могут содержать различные другие типы данных.

Как создаются и используются коллекции объектов

Для того чтобы создать коллекцию, необходимо описать тип данных, которые будут в ней храниться. Для простоты предположим некоторый абстрактный объект *TAbsObject*, который имеет конструктор *Init*.

Тогда создание коллекции, состоящей из элементов типа *TAbsObject*, будет выглядеть следующим образом:

```
Type
PAbsObject = ^TAbsObject;
TAbsObject = Object(TObject)
....
End;
```

```

Var
  AbsCollection : PCollection;

Begin
  AbsCollection := New(PCollection, Init(10,10));
  With AbsCollection^ do
    Begin
      {Поместить элементы в коллекцию}
      Insert(New(PAbsObject, Init(....)));
    End;
  End;
End;

```

Итак, мы посмотрели, как создается коллекция. При инициализации указывается начальный размер коллекции и шаг приращения, если число элементов превысит начальный размер. Так как в коллекции хранятся указатели, в ней могут храниться данные любого типа (определяемые через указатели).

Для работы с коллекциями помимо методов *Insert* (поместить) и *Delete* (удалить) используются специальные методы, называемые итераторами. Таких методов три: *FirstThat*, *ForEach* и *LastThat*. Каждый из методов-итераторов ожидает в качестве параметра указатель на процедуру типа *far*, которая и выполняет определенные действия.

Метод *FirstThat* вызывает указанную в качестве параметра функцию *Test*, которая выполняет проверку элемента по определенному критерию. Метод *FirstThat* перебирает все элементы коллекции и передает их функции *Test*. Метод *ForEach* позволяет выполнить определенные действия над всеми элементами коллекции. Отметим, что функция, указатель на которую передается в качестве параметра, должна иметь атрибут *far* и должна быть локальной.

Отсортированные и неотсортированные коллекции

Объект *TCollection*, на базе которого создаются коллекции в Turbo Vision, не имеет средств для сортировки. При необходимости получения отсортированной коллекции используется объект *TSortedCollection*. У этого объекта появляется метод *Compare*, который используется для сравнения двух указателей. Остальные методы, связанные с помещением и извлечением элементов коллекции, используют этот метод. Метод *TSortedCollection.Compare* является абстрактным методом: он должен быть переопределен.

Например, для коллекций, состоящих из строк, метод *Compare* может выглядеть следующим образом:

```
If PString(Key1)^ < PString(Key2)^ Then
  Compare := -1
Else If PString(Key1)^ < PString(Key2)^ Then
  Compare := +1
Else
  Compare := 0
```

Для сортировки в обратном порядке, необходимо поменять местами -1 и +1.

Сортировка без проверки на регистр (верхний или нижний) требует дополнительных действий. Как известно, функция *UpCase* правильно работает только с символами из первой половины таблицы *ASCII*. Сначала нам необходимо создать функцию, назовем ее *UpCaseC*, которая бы правильно преобразовывала символы кириллицы:

```
Function UpCaseC(Ch : Char) : Char;
Begin
  If Ch in ['a'..'z']
    Then UpCaseC := UpCase(Ch)
  Else If Ch in ['a'..'n']
    Then UpCaseC := Chr(Ord(Ch)-32)
  Else If Ch in ['p'..'я'] Then
    UpCaseC := Chr(Ord(Ch)-80)
End;
```

Далее нам необходимо создать функцию преобразования строки в верхний регистр:

```
Function ToUpperStr(P : PString) : String;
Var
  I : Integer;
  S : String;
Begin
  For I := 1 to Length(P^) do S[I] := UpCaseC(P^[I]);
  S[0] := P^[0];
  ToUpperStr := S;
End;
```

Тогда метод *Compare*, выполняющий сравнение, не зависящее от регистра, будет выглядеть следующим образом:

```
Function TSomeCollection.Compare(Key1, Key2 : Pointer) : Integer;
Var
  S1, S2 : String;
Begin
  S1 := ToUpperStr(PString(Key1));
  S2 := ToUpperStr(PString(Key2));
  If S1 < S2 Then Compare := -1 Else
```

```

If S1 > S2 Then Compare := +1 Else
    Compare := 0
End;

```

Для создания коллекций, состоящих из *ASCII*-строк, можно использовать объект *TStringCollection*. Элементами этой коллекции являются указатели на строки. Так как предком этого объекта является объект *TSortedCollection*, строки хранятся в отсортированном порядке. Для того, чтобы создать коллекцию строк, которые будут размещаться в ней в порядке поступления (и не будут сортироваться), необходимо создать объект-наследник объекта *TStringCollection*. Назовем этот объект *TUStringCollection*:

```

/////////////////////////////////////////////////////////////////
USTRCOLL.PAS - Пример создания объекта
TUStringCollection - коллекции строк, в которой
не производится сортировка
/////////////////////////////////////////////////////////////////
TYPE
    PUStringCollection = ^TUStringCollection;
    TUStringCollection = Object(TStringCollection)
        Procedure Insert(Item:Pointer);Virtual;
    End;

    Procedure TUStringCollection.Insert(Item:Pointer);
    begin
        AtInsert(Count,Item);
    End;

```

В данной реализации мы заменяем метод *Insert* на метод *AtInsert*, который помещает элемент в коллекцию по указанному индексу. Так как переменная *Count* содержит счетчик числа элементов коллекции, указанный элемент будет помещен в коллекцию последним. Таким образом, нам не нужно вызывать метода *Compare* и мы получаем коллекцию строк, в которой сортировка не производится.

Как коллекции используются отображаемыми объектами

Коллекции широко используются отображаемыми объектами Turbo Vision. Например, строки в объекте *TCluster* хранятся в коллекции типа *TStringCollection*, коллекция типа *TCollection* используется для отображения строк в списке (объект *TListBox*), и так далее.

Специальным типом коллекции является объект *TResourceCollection* (наследник объекта *TStringCollection*),

который используется для создания коллекций ресурсов. Этот объект используется объектом *TResourceFile*, который предназначен для создания и использования файлов-ресурсов. Более подробно об этих объектах рассказывается в разделе "Ресурсы".

Коллекции и требования к памяти

Размер коллекции может динамически изменяться, начиная с размера, указанного в конструкторе *Init* с шагом, указанным при инициализации коллекции. Внутри коллекции используется массив указателей размером в *MaxCollectionSize* элементов типа *Pointer*. Константа *MaxCollectionSize* задана следующим образом:

```
MaxCollectionSize = 65530 DIV SizeOf(Pointer);
```

Таким образом, учитывая, что размер указателя равен 4 байтам, мы получаем, что число элементов коллекции не может превышать 16380 элементов.

Если происходит переполнение коллекции, то вызывается метод *TCollection.Error*, что приводит к возникновению ошибки времени выполнения (*run-time error*). Для обработки ошибок такого типа необходимо переопределить данный метод.

В случае возникновения ошибки метод *Error* может возвращать два кода ошибки: *colIndexError* (-1) и *coOverflow* (-2). Код *colIndexError* указывает на то, что указан неверный индекс, а код *coOverflow* - на переполнение коллекции. Параметр *Info* может содержать дополнительную информацию. В случае неверного индекса параметр *Info* содержит значение индекса, а в случае переполнения - требуемый размер для расширения коллекции.

Потоки

Потоки - это средство для хранения объектов. Потоки, реализованные в Turbo Vision, позволяют хранить объекты в файлах и в памяти. Таким образом, используя потоки, можно говорить об объектно-ориентированном вводе/выводе. Как и в случае с коллекциями, в потоках могут храниться любые типы данных, не только объекты - потоки являются полиморфными. Любой отображаемый объект, реализованный в Turbo Vision, имеет два метода: *Put* - для сохранения объекта в потоке и *Get* - для

извлечения объекта из потока. При использовании потоков очень важно понимание концепции регистрации. Регистрация объекта состоит из двух шагов. Сначала создается специальная запись типа *TStreamRec*, которая содержит следующие поля:

```
TStreamRec = Record
  ObjType : Word;
  VMTLink : Word;
  Load   : Pointer;
  Store   : Pointer;
End
```

Поле *ObjType* содержит уникальный идентификатор типа объекта. Для нестандартных объектов можно использовать номера 1000 - 65535. В приложении содержится список идентификаторов стандартных объектов Turbo Vision. Поле *VMTLink* содержит указатель на таблицу виртуальных методов данного объекта. Поля *Load* и *Store* содержат указатели на методы *Load* и *Store* данного объекта.

Например, для объекта *TSomeObject*, регистрационная запись может выглядеть следующим образом:

```
RSomeObject : TStreamRec = (
  ObjType : 2000;
  VMTLink : OfS(TypeOf(TSomeObject));
  Load   : @TSomeObject.Load;
  Store   : @TSomeObject.Store;
);
```

После того как регистрационная запись создана, объект регистрируется с помощью функции *RegisterType*. После этого можно использовать объект совместно с потоками.

Стандартные объекты, входящие в состав Turbo Vision, имеют регистрационные записи и регистрация этих объектов выполняется соответствующими процедурами, которые перечислены в приведенной ниже таблице.

Процедура	Регистрирует объекты
RegisterApp	TBackground TDesktop
RegisterColorSel	TColorSelector TMonoSelector TColorDisplay TColorGroupList TColorItemList

RegisterDialogs	TColorDialog
	TDialog
	TInputLine
	TButton
	TCluster
	TRadioButton
	TCheckBoxes
	TMultiCheckBoxes
	TListBox
	TStaticText
	TLabel
	THistory
	TParamText
RegisterEditors	TEditor
	TMemo
	TFileEditor
	TIndicator
RegisterMenus	TEditWindow
	TMenuBar
	TMenuBox
	TStatusLine
RegisterStdDlg	TMenuPopup
	TFileInputLine
	TFileCollection
	TFileList
	TFileInfoPane
	TFileDialog
	TDireCollection
	TDireListBox
	TSortedListBox
	TChDireDialog
RegisterViews	TView
	TFrame
	TScrollBar
	TScroller
	TListViewer
	TGroup
	TWindow

Методы Load и Store

Для успешного создания этих методов необходимо руководствоваться следующими несложными правилами:

- сначала вызывается метод *Load* или *Store* объекта-предка данного объекта;

- затем сохраняется информация, необходимая для нормального функционирования объекта.

Рассмотрим небольшой пример из Turbo Vision. Объект *TWindow* является наследником объекта *TGroup*. Поэтому в методе *Store* первой строкой будет выполняться сохранение именно этого объекта:

```
TGroup.Store(S)
```

Затем необходимо сохранить флаги, размер прямоугольной области, номер окна, палитру, указатель на объект типа *TFrame* и заголовок окна:

```
S.Write(Flags, SizeOf(Byte) + SizeOf(TRect) + 2*SizeOf(Integer));
```

Мы сохранили значения полей, начиная с *Flags* и заканчивая *Palette*. Для сохранения еще одного отображаемого объекта используется метод *PutSubViewPtr*. Затем сохраняется заголовок окна:

```
S.WriteStr(Title)
```

Загрузка объекта из потока происходит в том же порядке: сначала считывается объект-предок, а затем поля самого объекта.

Изменения в Turbo Vision 2.0

В объекте *TStream* введены два новых метода - *StrRead* и *StrWrite*, поддерживающие чтение из потока и сохранение в потоке *ASCIZ*-строк.

Реализован новый объект *TMemoryStream*, представляющий собой поток, располагаемый в памяти. Для этого объекта реализованы все стандартные методы для работы с потоками: чтение из потока, запись в поток, получение размера потока и определение текущей позиции в потоке.

Переменная StreamError

Переменная *StreamError* (типа *Pointer*) предназначена для установки собственных обработчиков ошибок, возникающих при использовании потоков. По умолчанию

значение этой переменной равно *nil*. Переопределим значение этой переменной:

```
StreamError := @HandleStreamErr,
```

где процедура *HandleStreamError* объявлена следующим образом:

```
procedure HandleStreamErr( var S : TStream); far;
```

тогда эта процедура будет получать управление при возникновении ошибок, связанных с использованием потоков. Как реализовать такую процедуру, показано ниже.

```
{//////////////////////////////////////
STRM_ERR.PAS: Переопределение переменной StreamError
//////////////////////////////////////}
uses Objects;
Type
  PSafeStream = ^TSafeStream;
  TSafeStream = Object(TBufStream)
    procedure Error(Code, Info : Integer);    virtual;
  End;

Procedure HandleStreamError(Var Str : TStream); FAR;
Begin
  Case Str.Status of
    stError      : Writeln('Access Error');
    stInitError  : Writeln('Init Error');
    stReadError  : Writeln('Read Error');
    stWriteError : Writeln('Write Error');
    stGetError   : Writeln('Get Error');
    stPutError   : Writeln('Put Error');
  End;
End;

Procedure TSafeStream.Error;
Begin
  Status := Code;
  ErrorInfo := Info;
  HandleStreamError(Self);
End;

Var
  S : TSafeStream;
  W : Word;
Begin
  StreamError := @HandleStreamError;
  S.Init('DEMO.BIN', stOpen, 1024);
  S.Reset;
  S.Read(W, SizeOf(W));
End.
```

Отметим, что помимо реализации процедуры *HandleStreamError* также необходимо создать объект -

наследник объекта, реализующего необходимый тип потока (*TBufStream*, *TDosStream*, *TEMSStream* и т.д.) и переопределить метод *Error*. Параметр *Str* типа *Stream* дает процедуре *HandleStreamError* доступ к полям экземпляра объекта, реализующего поток. Поле *Status* содержит код ошибки, а поле *ErrorInfo* - дополнительную информацию, в зависимости от типа ошибки. Так, для ошибок *stError*, *stInitError*, *stReadError* и *stWriteError* это поле будет содержать код ошибки *DOS*, для ошибки *stGetError* - идентификатор объекта (значение поля *ObjType* записи *TStreamRec*), а для ошибки *stPutError* это поле будет содержать значение поля *VmtLink* записи *TStreamRec*. Для отмены ошибочного состояния и продолжения работы с потоком необходимо вызвать метод *Reset*. Этот метод обнуляет значения полей *Status* и *ErrorInfo*.

Объект TMemoryStream

Объект *TMemoryStream* реализует поток в памяти. Такие потоки удобно использовать, в первую очередь, в защищенном режиме, где мы можем выделить достаточное количество памяти. Использование потоков в памяти является крайне простым. Сначала поток создается с помощью конструктора - указывается начальный размер потока и размер блока. После того как поток создан, он может использоваться как обычный дисковый поток. Запись за концом потока вызывает увеличение размера блока. При использовании потоков в памяти необходимо помнить, что изменение размеров потока может привести к фрагментации кучи. Для того чтобы избежать таких эффектов, необходимо подобрать оптимальные значения для начального размера и размера самого блока.

Пути расширения потоков

Потоки, как одно из наиболее удобных средств для работы с объектами, могут быть расширены для придания им еще большей гибкости. Так, например, в Turbo Vision версии 2.0 был введен новый тип потока - *TMemoryStream*, который располагается в памяти. Изучив реализацию стандартных потоков, можно создать поток, располагающийся в дополнительной памяти - *TXMSStream* (по аналогии с *TEMSStream*), а также расширения файловых

потоков, например, потоки с шифрованием или потоки со сжатием информации.

Ресурсы

Концепция ресурсов - интерфейсных элементов, хранимых отдельно от кода программы, возможно, в отдельном файле, аналогична ресурсам, используемым в среде Microsoft Windows. В отличие от Windows, не поставляется стандартный редактор ресурсов, но все необходимые для его реализации средства присутствуют в Turbo Vision. Использование ресурсов позволяет легко изменять внешний вид приложения без модификации самого кода. Как говорится в документации, использование ресурсов может сократить размер кода вашей программы до 10%. Также использование ресурсов может быть полезно при создании локальных (или многоязычных) версий приложений. Еще одно применение ресурсов - создание версий программ с усеченными возможностями, т.е. демонстрационных версий. В этом случае вы создаете законченную программу, но располагаете в файле ресурсов только ограниченный набор интерфейсных элементов. Для превращения демо-версии программы в функциональную версию необходимо только изменить содержимое файла ресурсов.

Создание файла ресурсов

Файл ресурсов - это поток произвольного доступа, в котором содержатся объекты, каждый из которых доступен по уникальному "ключу" - строке, идентифицирующей объект. Обычно для создания файла ресурсов используется специальная версия программы, которая содержит в себе только процедуры создания интерфейсных элементов, а также процедуры для сохранения этих элементов в ресурсе. В приведенном примере показано создание меню, которое сохраняется в файле ресурсов с "ключом" *MainMenu*.

```
{//////////////////////////////////////  
RES_OUT.PAS: Построение интерфейсных элементов и  
сохранение их в виде ресурса  
//////////////////////////////////////}  
uses Drivers, Objects, Views, App, Menus, Dialogs;  
Const  
  cmFileOpen = 300;  
  cmFileNew = 301;
```

```

Var
  ResFile : TResourceFile;
  ResStrm : PBufStream;

Procedure MakeMenuBar;
Var
  MenuBar : PMenuBar;
  R      : TRect;
Begin
  R.Assign(0, 0, 80, 1);
  MenuBar := New(PMenuBar, Init(R, NewMenu(
    NewSubMenu('~F~ile', hcNoContext, NewMenu(
      NewItem('~O~pen', 'F3', kbF3, cmFileOpen, hcNoContext,
      NewItem('~N~ew ', 'F4', kbF4, cmFileNew, hcNoContext,
      NewLine(
        NewItem('E~x~it', 'Alt-X', kbAltX, cmQuit, hcNoContext,
        nil))))),
    NewSubMenu('~W~indow', hcNoContext, NewMenu(
      NewItem('~N~ext', 'F6', kbF6, cmNext, hcNoContext,
      NewItem('~Z~oom', 'F5', kbF5, cmZoom, hcNoContext,
      nil))),
    nil)))
  ));
  ResFile.Put(MenuBar, 'MainMenu');
  Dispose(MenuBar, Done);
End;

Begin

  RegisterObjects;
  RegisterMenus;

  ResStrm := New(PBufStream, Init('DEMO.RES', stCreate, 512));
  If ResStrm^.Status <> stOk Then Writeln('Stream error');
  ResFile.Init(ResStrm);
  MakeMenuBar;
  ResFile.Done;
End.

```

Примечание: сначала создается поток (*ResStrm*), который будет использоваться с файлом ресурсов. Затем после успешного создания потока происходит инициализация файла ресурсов. Обратите внимание, что нам необходимо зарегистрировать те объекты, которые будут использоваться в потоках. Это является важным условием правильной работы с файлами ресурсов.

Использование файла ресурсов

После того как файл ресурсов создан, его можно использовать для инициализации интерфейсных объектов. Как это происходит показано ниже:

```

/////////////////////////////////////////////////////////////////
RES_IN.PAS: Пример использования файла ресурсов
/////////////////////////////////////////////////////////////////
uses Objects, Drivers, Menus, Views, Dialogs, App;

Var
  ResFile : TResourceFile;

Type
  TDemoApp = Object(TApplication)
    constructor Init;
    procedure InitMenuBar;      virtual;
  End;

Constructor TDemoApp.Init;
Var
  ResStrm : PBufStream;
Begin
  RegisterObjects;
  RegisterMenus;

  ResStrm := New(PBufStream, Init('DEMO.RES', stOpenRead, 512));
  If ResStrm^.Status <> stOk Then Writeln('Stream error');
  ResFile.Init(ResStrm);

  Inherited Init;
End;

Procedure TDemoApp.InitMenuBar;
Begin
  MenuBar := PMenuBar(ResFile.Get('MainMenu'));
End;

Var
  DemoApp : TDemoApp;
Begin
  DemoApp.Init;
  DemoApp.Run;
  DemoApp.Done;
End.

```

Примечание: единственным отличием данной программы от обычной является то, что в методе *InitMenuBar* вместо стандартной инициализации полосы меню, мы загружаем ее из ресурса. Необходимые манипуляции с потоком и файлом ресурсов происходят в конструкторе *Init*. Еще раз обратим внимание на необходимость регистрации тех объектов, которые будут использоваться в потоках:

```

      RegisterObjects;
      RegisterMenus;

```

Если не планируется дальнейшее изменение интерфейса приложения, то файл ресурсов можно присоединить к приложению:

```
REN DEMO.EXE DEMO.BIN
COPY /B DEMO.BIN + DEMO.RES DEMO.EXE
```

Также в конструкторе Init необходимо изменить следующую строку:

```
ResStrm :=
New(PBufStream, Init('DEMO.RES', stOpenRead, 512));
```

на строку:

```
ResStrm :=
New(PBufStream, Init(ParamStr(1), stOpenRead, 512));
```

Если предполагается создание локальных версий прикладной программы, то имеет смысл расположить ресурсы в отдельном файле. В этом случае для изменения языка приложения потребуется всего лишь замена файла ресурсов.

Приведенная выше методика распространяется на все интерфейсные объекты, реализованные в Turbo Vision. Например, для сохранения панели диалога в файле ресурсов и ее последующего использования, необходимо выполнить следующие действия:

```
R.Assign(0,0,40,15);
MainDialog := New(PDialogBox, Init(R, 'Mail Dialog'));
With MainDialog do
Begin
.....

R.Assign(20,8,30,10);
Insert(New(PButton, Init(R, '~O~k', cmOk, bfDefault)));
.....
End;

ResFile.Put(MainDialog, 'MainDialog');
```

Загрузка такой панели диалога будет выглядеть следующим образом:

```
MainDialog := PDialog(ResFile.Get('MainDialog'));
```

Более удобной является прямое отображение панели диалога:

```
ExecuteDialog(PDialog(ResFile.Get('MainDialog')), Nil);
```

Как и во всех случаях использования объектов в потоках, необходимо выполнить регистрацию. В данном случае нужно вызвать процедуры *RegisterDialogs* и *RegisterViews*.

Метод *TResourceFile.SwitchTo*

Этот метод может использоваться для "упаковки" файла с ресурсами. Если параметр *Pack* равен *True*, то при копировании из одного файла с ресурсами в другой указатели типа *Nil* не записываются. Если параметр *Pack* равен *False*, выполняется обычное копирование.

Доступ к ресурсам в файлах

В этом разделе мы рассмотрим, как с помощью объектов *Turbo Vision* осуществляется доступ к ресурсам, которые хранятся либо в отдельных файлах, либо просоединены к исполняемым файлам. В качестве примера рассмотрим чисто ресурсный файл, хотя описываемая методика распространяется также и на исполняемый файл с ресурсами.

Для того чтобы найти ресурсы в файле (назовем его *DEMO.RES*), нам необходимо открыть его как ресурсный файл:

```
ResFile.Init(New  
  (PBufStream, Init('DEMO.RES', stOpenRead, 2048)));
```

Поле *ResFile.Count* укажет нам число ресурсов, хранимых в данном файле. Далее нам необходимо перебрать все ресурсы и, например, после того как ресурс данного типа найден, загрузить его или обработать соответствующим образом. Для этого мы в цикле загружаем каждый ресурс и проверяем его тип:

```
Obj := ResFile.Get(ResFile.KeyAt(Count));  
If TypeOf(Obj) = TypeOf(TDialog) ...
```

После того как необходимый тип ресурса найден, мы выполняем процедуру его обработки. Так как объект уже загружен (объект *Obj*), мы можем преобразовать его в необходимый тип, например:

```
PDIALOG(Obj)
```

и выполнить над ним любые действия, все поля данного объекта будут нам доступны.

В приведенной ниже программе показано, как получить список стандартных ресурсов, хранимых в файле ресурсов.

```
{//////////////////////////////////////
LIST_RES: Показ списка ресурсов в файле
//////////////////////////////////////}
uses Objects, Dialogs, Views, Menus, App, TextView, Memory, HistList;
Var
  ResFile : TResourceFile;
  Count   : Integer;
  Obj     : PObject;

Begin
  RegisterObjects;
  RegisterViews;
  RegisterDialogs;
  RegisterMenus;
  RegisterApp;
  RegisterType(RStringList);

  ResFile.Init(New(PBufStream, Init(ParamStr(1), stOpenRead, 2048)));
  Writeln('Resources in file = ', ResFile.Count);
  For Count := 0 to ResFile.Count-1 do
    Begin
      Obj := ResFile.Get(ResFile.KeyAt(Count));
      If TypeOf(Obj) = TypeOf(TDialog) Then Writeln('Dialog');
      If TypeOf(Obj) = TypeOf(TMenuBar) Then Writeln('Menu');
      If TypeOf(Obj) = TypeOf(TStringList) Then Writeln('String List');
    End;
  End.
```

Таким образом, если мы умеем добираться до ресурсов, хранимых в файлах, мы можем их изменять. По этому принципу создаются редакторы ресурсов.

Для замены одного объекта на другой, скажем, при создании локальной версии программы, когда файл ресурсов уже подключен к исполняемому файлу, можно использовать комбинацию методов *Delete* и *Put*. Сначала мы удаляем объект с указанным "ключом":

```
ResFile.Delete('MainMenu');
```

а затем помещаем в файл ресурсов новый объект с тем же "ключом":

```
ResFile.Put('MainMenu');
```

Отметим, что в этом случае файл ресурсов должен быть открыт для записи:

```
ResFile.Init(New(
```

```
PBufStream,Init(ParamStr(1),stOpen, 2048)));
```

Используя описанные выше методы, можно создать утилиту, которая автоматически изменяет содержимое файла ресурсов.

Завершая рассмотрение файлов ресурсов, хочу отметить, что использование ресурсов для хранения интерфейсных элементов поможет сделать перенос программ, созданных с помощью Turbo Vision, в среду Microsoft Windows более простым. Некоторые вопросы переноса программ рассматриваются в следующей главе.

Объекты TStringList и TStrListMaker

Объект *TStringList* представляет собой механизм для доступа к строкам, хранящимся в потоках. Каждая строка в этом случае идентифицируется по "ключу" - уникальному числу в диапазоне от 0 до 65535. Использование этого объекта вместо хранения строк непосредственно в памяти существенно сокращает размер памяти, занимаемой программой, и позволяет довольно простым способом выполнять локализацию программ. Потоки, содержащие строки, создаются с помощью объекта *TStrListMaker*. Как создается такой поток, показано в следующем примере.

```
/////////////////////////////////////////////////////////////////
STRLIST.PAS: Пример создания потока со строками
/////////////////////////////////////////////////////////////////
uses Objects;
Const
  sLine0 = 0; sLine1 = 1;
  sLine2 = 2; sLine3 = 3;
  sLine4 = 4; sLine5 = 5;
Var
  ResFile : TResourceFile;
  StrList : TStrListMaker;
Begin
  RegisterType(RStrListMaker);
  ResFile.Init(New(PBufStream, Init('DEMO.RES', stCreate, 2048)));
  StrList.Init(4096, 32);

  StrList.Put(sLine0, 'Line #0');
  StrList.Put(sLine1, 'Line #1');
  StrList.Put(sLine2, 'Line #2');
  StrList.Put(sLine3, 'Line #3');
  StrList.Put(sLine4, 'Line #4');
  StrList.Put(sLine5, 'Line #5');

  ResFile.Put(@StrList, 'Lines');
  StrList.Done;
  ResFile.Done;
```

End.

Необходимо заметить, что в конструкторе *StrList.Init* указывается два параметра, первый из которых указывает размер буфера, отводимого для хранения строк. Буфер должен быть достаточно большим, чтобы вместить все строки. Каждая строка занимает на один байт больше числа символов в ней. Второй параметр конструктора указывает число элементов в буфере. При добавлении строк в список строится индекс. Строки с последовательными "ключами" сохраняются в виде специальных записей, по 16 строк каждая.

После того как поток создан, возможно его использование в прикладной программе, как это показано в следующем примере.

```
/////////////////////////////////////////////////////////////////
STRUSE.PAS Пример использования потока строк,
созданного программой STRLIST.PAS
/////////////////////////////////////////////////////////////////
uses Objects;
Const
  sLine0 = 0; sLine1 = 1;
  sLine2 = 2; sLine3 = 3;
  sLine4 = 4; sLine5 = 5;
Var
  ResFile : TResourceFile;
  StrList : PStringList;
  Count : Byte;
Begin
  RegisterType(RStringList);
  ResFile.Init(New(PBufStream, Init('DEMO.RES', stOpenRead, 2048)));
  StrList := PStringList(ResFile.Get('Lines'));
  For Count := sLine0 to sLine5 do
    Writeln('Line #', Count, ' = ', StrList.Get(Count));
End.
```

Необходимо помнить, что для успешного использования объектов *TStrListMaker* и *TStringList* их необходимо зарегистрировать перед использованием в потоках. Для этого используется процедура *RegisterType*.

При необходимости созданный указанным выше способом поток можно присоединить непосредственно к исполняемому файлу. Для этого необходимо выполнить следующие команды:

```
REN DEMO.EXE DEMO.BIN
COPY /B DEMO.BIN + DEMO.RES DEMO.EXE
```

и изменить вызов конструктора таким образом:

```
ResFile.Init(New(PBufStream, Init(ParamStr(0), stOpenRead, 2048)));
```

Оба объекта имеют одинаковый регистрационный номер и не могут использоваться в одной программе. Таким образом, поток, содержащий строки, должен создаваться одной программой (содержащей объект *TStrListMaker*), тогда как работа с этим потоком должна осуществляться другой программой (содержащей объект *TStringList*). Такой подход не всегда является удобным. Чтобы это ограничение не служило препятствием, необходимо выполнить следующие действия: после того как поток, содержащий строки, создан, нам больше не нужен объект *TStrListMaker*. Мы можем отменить его регистрацию, присвоив полю *ObjType* записи *RStrListMaker* заведомо неиспользуемое значение:

```
RStrListMaker.ObjType := Word(-1);
```

и после этого зарегистрировать объект *TStringList*:

```
RegisterType(RStringList);
```

Заключение

Неотображаемые объекты, входящие в состав Turbo Vision, представляют собой мощное средство управления данными. Например, известны случаи использования коллекций для создания баз данных. Объекты, реализованные в модуле *OBJECTS*, являются средо-независимыми: их можно использовать как в программах, создаваемых на базе библиотеки Turbo Vision, так и в Windows-программах. Это позволяет создавать более гибкие программы, совместимые по формату хранящихся данных.

ПРИЛОЖЕНИЕ А

Регистрационные коды

В приведенной ниже таблице показаны регистрационные коды стандартных объектов, используемые при регистрации этих объектов в потоках.

Объект	Код
TView	1
TFrame	2
TScrollBar	3
TScroller	4
TListViewer	5
TGroup	6
TWindow	7
TDialog	10
TInputLine	11
TButton	12
TCluster	13
TRadioButtons	14
TCheckBoxes	15
TListBox	16
TStaticText	17
TLabel	18
THistory	19
TParamText	20
TColorSelector	21
TMonoSelector	22
TColorDisplay	23
TColorGroupList	24
TColorItemList	25
TColorDialog	26
TMultiCheckBoxes	27
TBackground	30
TDeskTop	31
TMenuBar	40
TMenuBox	41
TStatusLine	42
TMenuPopup	42
TFileInputLine	60

TFileCollection	61
TFileList	62
TFileInfoPane	63
TFileDialog	64
TDirCollection	65
TDirListBox	66
TChDirDialog	67
TSortedListBox	68
TEditor	70
TMemo	71
TFileEditor	72
TIndicator	73
TEditWindow	74
TOutLine	91

Реакция стандартных объектов на события

Ниже приводится описание реакций стандартных объектов на события. Эти реакции реализованы в методах `HandleEvent` соответствующих объектов.

Объект TApplication

Сначала пришедшее событие отдается на обработку объекту `TProgram`. Обрабатываются команды: `cmTile` - вызывается метод `Tile`, `cmCascade` - вызывается метод `Cascade`, `cmDosShell` - вызывается метод `DosShell`.

Объект TButton

При нажатии одним из трех способов: с помощью мыши (`evMouseDown`), нажатии командной клавиши (`evKeyDown`) или по получении сообщения `cmDefault`, - происходит посылка присвоенной кнопке команды с помощью метода `TView.PutEvent`.

Объект TCluster

Вызывает метод `TView.HandleEvent` для обработки всех сообщений от мыши и клавиатуры. Элементы управления,

выбираемые мышью или нажатием клавиши на клавиатуре, отображаются соответствующим образом.

Объект TColorDialog

Обрабатывается команда `cmNewColorItem`, в результате которой выбирается новый индекс группы и команда `cmNewColorIndex`.

Объект TColorDisplay

Обрабатываются команды (`Event.Command`): `cmColorBackgroundChanged` и `cmColorForegroundChanged`, в результате которых происходит установка цвета и перерисовка отображаемого объекта (с помощью метода `DrawView`).

Объект TColorGroupList

Обрабатывается команда `cmSaveColorIndex`, в результате которой вызывается метод `SetGroupIndex`.

Объект TColorItemList

Обрабатывается команда (`Event.Command`) : `cmNewColorItem`, по которой фокус устанавливается на новый элемент `FocusItem(Group.Index)` и происходит перерисовка отображаемого объекта (`DrawView`).

Объект TColorSelector

Вызывает метод `TView.HandleEvent`. При нажатии кнопки мыши (событие `evMouseDown`) происходит выбор определенного цвета, который присваивается переменной `Color`. При нажатии клавиши (событие `evKeyDown`) происходит проверка области цветов (фон или цвет текста). После этого цвет выбирается клавишами "стрелка влево", "стрелка вправо", "стрелка вверх" и "стрелка вниз".

Передаёт событие на обработку объекту TGroup. Если событие является командой (Event.What = evCommand), то выполняется следующая проверка: если получена команда cmNext (обычно в результате нажатия клавиши F6), вызывается метод FocusNext(False), если получена команда cmPrev, то выполняются следующие действия:

```
if Valid(cmReleasedFocus) then    Current^.PutInFrontOf(Background);
```

Объект TDesktop обрабатывает две команды: cmNext и cmPrev.

Объект TDialog

Вызывает метод TWindow.HandleEvent. При нажатии клавиши (Event.What = evKeyDown) проверяется, нажата ли клавиша Esc (kbEsc), и в этом случае в очередь событий помещается команда cmCancel. При нажатии клавиши Enter (kbEnter) в очередь событий помещается команда cmDefault. Объект TDialog также обрабатывает команды (Event.What = evCommand). При получении команд cmOk, cmCancel, cmYes, cmNo и модальном состоянии панели диалога её модальное состояние завершается:

```
EndModal(Event.Command)
```

Объект TEditor

Вызывает метод TView.HandleEvent. Затем выполняет преобразование события с помощью метода ConvertEvent, который преобразует нажатия клавиш в команды, которые затем обрабатываются.

Объект TEditWindow

Обрабатывается одна команда - cmUpdateTitle, в результате которой вызывается метод Frame^.DrawView.

Вызывается метод TView.HandleEvent(Event). Обработка остальных сообщений передается объектам, помещенным в группу:

```
ForEach(@DoHandleEvent);
```

Объект TProgram

Если полученное событие является нажатием клавиши (Event.What = evKeyDown), то проверяется, была ли нажата комбинация клавиш Alt и клавиши в диапазоне от 1 до 9, и объекту TDesktop посылается команда выбора окна с номером нажатой клавиши:

```
Message(Desktop, evBroadCast, cmSelectWindowNum, Pointer(Byte(C) - $30))
```

Остальные события передаются объекту TGroup. Если событие является командой, то проверяется, не является ли эта команда командой cmQuit. В случае соответствия завершается модальное состояние объекта:

```
EndModal(cmQuit);
```

Объект TProgram обрабатывает команду cmQuit и клавиатурные события от клавиш Alt1..Alt9.

Объект TInputLine

Вызывает метод TView.HandleEvent. Затем обрабатываются события от клавиатуры и мыши, если строка редактирования является выбранной. Этот метод реализует стандартные свойства редактирования строки ввода.

Объект TLabel

Вызывает метод TStaticText.HandleEvent. Если получено сообщение evMouseDown или нажата командная клавиша, выбирается элемент управления, связанный с этим объектом. При получении команд cmReceivedFocus и cmReleasedFocus

происходит изменение значения поля `Light` и перерисовка объекта с помощью метода `DrawView`.

Объект `TMonoSelector`

Все события обрабатываются методом `TCluster.HandleEvent`. Обрабатывается команда `cmColorSet`, в результате которой происходит перерисовка отображаемого объекта (с помощью метода `DrawView`).

Объект `THistoryViewer`

Если произошло двойное нажатие кнопки мыши: ((`Event.What = evMouseDown`) and (`Event.Double`)) или была нажата клавиша `Enter`: ((`Event.What = evKeyDown`) and (`Event.KeyCode = kbEnter`)), то происходит завершение модального состояния объекта. Если же была нажата клавиша `Esc`: ((`Event.What = evKeyDown`) and (`Event.KeyCode = kbEsc`)) или получена команда `cmCancel`: ((`Event.What = evCommand`) and (`Event.Command = cmCancel`)) то модальное состояние завершается с кодом `cmCancel`. Остальные события передаются на обработку объекту `TListViewer`.

Объект `THistory`

Вызывает метод `TView.HandleEvent` и обрабатывает два события: при нажатии клавиши "стрелка вниз" (`CtrlToArrow(Event.KeyCode) = kbDown`) или активации кнопки отображается список; при потере фокуса (`cmReleasedFocus`) возвращается выбранный элемент из списка `RecordHistory(Link^.Data^)`.

Объект `TMemo`

Все сообщения, кроме `evKeyDown` и нажатия клавиши `Tab`, передаются объекту-предку (`TEditor`).

При нажатии клавиши (evKeyDown) происходит поиск меню по командной клавише : FindItem(GetCtrlChar(Event.KeyCode)) и посылка команды ;:

```
Event.What := evCommand;  
Event.Command := P.Command;  
Event.InfoPtr := nil;  
PutEvent(Event);
```

Отсальные события обрабатываются методом HandleEvent объекта TMenuBox.

Объект TMenuView

При нажатии кнопки мыши (Event.What = evMouseDown) происходит выбор элемента меню. При нажатии клавиши (Event.What = evKeyDown), если код клавиши соответствует одному из элементов меню, т.е.

```
if (FindItem(GetAltChar(Event.KeyCode)) <> nil),
```

также происходит выбор элемента.

Объект TFileEditor

Обрабатываются две команды Event.What = evCommand. По команде cmSave вызывается метод Save, по команде cmSaveAs - метод SaveAs.

Объект TStatusLine

Вызывает метод TView.HandleEvent. Обрабатывается три специальных типа событий. При нажатии кнопки мыши (evMouseDown) на элементе строки состояния происходит посылка команды. При нажатии клавиши (evKeyDown), соответствующей одному из элементов строки состояния, также происходит посылка команды. При получении команды cmCommandSetChanged происходит перерисовка отображаемого объекта (DrawView).

Вызывает метод TScroller.HandleEvent. Обработываемые события от клавиатуры и мыши приводят к перемещению фокуса на определенный элемент и к выбору элемента. Также при нажатии клавиши "+" происходит отображение вложенной иерархии.

Объект TFileInputLine

Вызывает метод TInputLine.HandleEvent(Event).
Обработывает команду cmFileFocused - возвращает содержимое поля TSearchRec.Name.

Объект TSortedListBox

Большинство событий от клавиатуры и "мыши" передается на обработку методу TListBox.HandleEvent. Также обрабатываются клавиатурные события, связанные с инкрементным поиском: когда пользователь нажимает на символьную клавишу, фокус перемещается на элемент, начинающийся с этого символа, нажатие второй клавиши приводит к перемещению фокуса на элемент, второй символ которого эквивалентен нажатому, и т.д.

Объект TFileList

Если произошло двойное нажатие кнопки мыши, то посылается команда cmOk:

```
Event.What := evCommand;  
Event.Command := cmOK;  
PutEvent(Event);  
ClearEvent(Event);
```

Остальные события обрабатываются методом TSortedListBox.HandleEvent.

Объект TFileInfoPane

Обрабатывается команда cmFileFocused, в результате которой устанавливается значение переменной S типа PSearchRec, и выполняется перерисовка отображаемого объекта(DrawView).

Объект TFileDialog

Вызывается метод TDialog.HandleEvent. Обрабатываются команды cmFileOpen, cmFileReplace и cmFileClear: происходит завершение модального состояния панели диалога.

Объект TDirListBox

Если произошло двойное нажатие кнопки мыши, посылается команда cmChangeDir:

```
Event.What := evCommand;  
Event.Command := cmChangeDir;  
PutEvent(Event);  
ClearEvent(Event);
```

Остальные события обрабатываются методом TListBox.HandleEvent.

Объект TChDirDialog

Вызывается метод TDialog.HandleEvent(Event). Обрабатываются команды (Event.What = evCommand): cmRevert - вызывается метод GetDir и cmChangeDir - происходит отображение нового каталога.

Объект TView

Если произошло событие (evMouseDown) - нажатие кнопки мыши, то выполняются следующие действия: если объект не выбран (sfSelected) и не запрещен его выбор

(sfDisabled), а также, если объект может быть выбран (ofSelectable), то он выбирается с помощью метода Select.

Объект TFrame

Вызывает метод TView.HandleEvent. Затем обрабатываются события от мыши. Если произошло нажатие мыши на кнопке закрытия, то посылается команда cmClose:

```
Event.What := evCommand;  
Event.Command := cmClose;  
Event.InfoPtr := Owner;  
PutEvent(Event);  
ClearEvent(Event);
```

При нажатии мыши на кнопке максимизации окна посылается команда cmZoom:

```
Event.What := evCommand;  
Event.Command := cmZoom;  
Event.InfoPtr := Owner;  
PutEvent(Event);  
ClearEvent(Event);
```

При перемещении мыши с нажатой кнопкой происходит перемещение окна - вызывается метод DragWindow с параметром dmDragMove, при попытке изменения размеров окна - метод DragWindow с параметром dmDragGrow.

Объект TScrollBar

Вызывается метод TView.HandleEvent(Event). Если произошло нажатие кнопки мыши, оно передается владельцу полосы прокрутки (с помощью функции Message), который должен обрабатывать изменения положения бегунка. Также определяется часть полосы прокрутки, в которой произошло нажатие кнопки мыши. Значение поля Value изменяется в соответствии со значениями ArStep и PgStep.

Объект TScroller

Вызывается метод TView.HandleEvent. Получение команд cmScrollBarChanged с значением поля Event.InfoPtr равным

HScrollBar или VScrollBar приводит к вызову метода ScrollDraw.

Объект TListViewer

Вызывается метод TView.HandleEvent. При нажатии кнопки мыши перемещается фокус на определенный элемент списка. Выбор элемента осуществляется двойным нажатием кнопки мыши. Обрабатываются следующие клавиатурные события: нажатие "пробела" приводит к выбору элемента, клавиши-"стрелки" и клавиши PgUp, PgDn, Ctrl-PgDn, Ctrl-PgUp, Home и End приводят к перемещению фокуса на определенный элемент списка. События от полос прокрутки приводят к перемещению фокуса и перерисовке объекта с помощью метода DrawView.

Объект TWindow

Вызывает метод TGroup.HandleEvent. Обрабатываются команды cmResize - устанавливаются новые размеры окна (SizeLimits), а затем происходит перемещение отображаемого объекта (DragView); cmClose - посылается команда cmCancel; cmZoom - выполняется метод Zoom. Обрабатываемые клавиатурные события: нажатие клавиши Tab (kbTab) или Shift+Tab (kbShiftTab) - происходит установка фокуса на следующий или предыдущий объект внутри окна. Если получена команда cmSelectWindowNum - выбор окна по номеру, - происходит вызов метода Select.

ПРИЛОЖЕНИЕ Б

TURBO VISION 2.0

Расширенная иерархия объектов

—TObject	
Done	(Destruct; virtual)
Free	(Proc)
Init	(Construc)
—TStream	
CopyFrom	(Proc)
Error	(Proc; virtual)
ErrorInfo	(Field)
Flush	(Proc; virtual)
Get	(Function)
GetPos	(Function; virtual)
GetSize	(Function; virtual)
Init	(Construc)
Put	(Proc)
Read	(Proc; virtual)
ReadStr	(Function)
Reset	(Proc)
Seek	(Proc; virtual)
Status	(Field)
StrRead	(Function)
StrWrite	(Proc)
Truncate	(Proc; virtual)
Write	(Proc; virtual)
WriteStr	(Proc)
—TDosStream	
Done	(Destruct; virtual)
GetPos	(Function; virtual)
GetSize	(Function; virtual)
Handle	(Field)
Init	(Construc)
Read	(Proc; virtual)
Seek	(Proc; virtual)
Truncate	(Proc; virtual)
Write	(Proc; virtual)
—TBufStream	
BufEnd	(Field)
Buffer	(Field)
BufPtr	(Field)
BufSize	(Field)
Done	(Destruct; virtual)
Flush	(Proc; virtual)
GetPos	(Function; virtual)

GetSize	(Function; virtual)
Init	(Construc)
Read	(Proc; virtual)
Seek	(Proc; virtual)
Truncate	(Proc; virtual)
Write	(Proc; virtual)
TEmsStream	
Done	(Destruct; virtual)
GetPos	(Function; virtual)
GetSize	(Function; virtual)
Handle	(Field)
Init	(Construc)
PageCount	(Field)
Position	(Field)
Read	(Proc; virtual)
Seek	(Proc; virtual)
Size	(Field)
Truncate	(Proc; virtual)
Write	(Proc; virtual)
TMemoryStream	
BlockSize (Field)	
ChangeListSize	(Function; private)
CurSeg	(Field)
Done	(Destruct; virtual)
GetPos	(Function; virtual)
GetSize	(Function; virtual)
Init	(Construc)
Position	(Field)
Read	(Proc; virtual)
Seek	(Proc; virtual)
SegCount	(Field)
SegList	(Field)
Size	(Field)
Truncate	(Proc; virtual)
Write	(Proc; virtual)
TCollection	
At	(Function)
AtDelete	(Proc)
AtFree	(Proc)
AtInsert	(Proc)
AtPut	(Proc)
Count	(Field)
Delete	(Proc)
DeleteAll	(Proc)
Delta	(Field)
Done	(Destruct; virtual)
Error	(Proc; virtual)
FirstThat	(Function)
ForEach	(Proc)
Free	(Proc)

FreeAll	(Proc)
FreeItem	(Proc; virtual)
GetItem	(Function; virtual)
IndexOf	(Function; virtual)
Init	(Construc)
Insert	(Proc; virtual)
Items	(Field)
LastThat	(Function)
Limit	(Field)
Load	(Construc)
Pack	(Proc)
PutItem	(Proc; virtual)
SetLimit	(Proc; virtual)
Store	(Proc)
TSortedCollection	
Compare	(Function; virtual)
Duplicates	(Field)
IndexOf	(Function; virtual)
Init	(Construc)
Insert	(Proc; virtual)
KeyOf	(Function; virtual)
Load	(Construc)
Search	(Function; virtual)
Store	(Proc)
TStringCollection	
Compare	(Function; virtual)
FreeItem	(Proc; virtual)
GetItem	(Function; virtual)
PutItem	(Proc; virtual)
TResourceCollection	
FreeItem	(Proc; virtual)
GetItem	(Function; virtual)
KeyOf	(Function; virtual)
PutItem	(Proc; virtual)
TStrCollection	
Compare	(Function; virtual)
FreeItem	(Proc; virtual)
GetItem	(Function; virtual)
PutItem	(Proc; virtual)
TFileCollection	
Compare	(Function; virtual)
FreeItem	(Proc; virtual)
GetItem	(Function; virtual)
PutItem	(Proc; virtual)
TDirCollection	
FreeItem	(Proc; virtual)
GetItem	(Function; virtual)
PutItem	(Proc; virtual)

—TResourceFile	
BasePos	(Field; private)
Count	(Function)
Delete	(Proc)
Done	(Destruct; virtual)
Flush	(Proc)
Get	(Function)
Index	(Field; private)
IndexPos	(Field; private)
Init	(Construc)
KeyAt	(Function)
Modified	(Field)
Put	(Proc)
Stream	(Field)
SwitchTo	(Function)
—TStringList	
BasePos	(Field; private)
Done	(Destruct; virtual)
Get	(Function)
Index	(Field; private)
IndexSize	(Field; private)
Load	(Construc)
ReadStr	(Proc; private)
Stream	(Field; private)
—TStrListMaker	
CloseCurrent	(Proc; private)
Cur	(Field; private)
Done	(Destruct; virtual)
Index	(Field; private)
IndexPos	(Field; private)
IndexSize	(Field; private)
Init	(Construc)
Put	(Proc)
Store	(Proc)
Strings	(Field; private)
StrPos	(Field; private)
StrSize	(Field; private)
—TView	
Awaken	(Proc; virtual)
BlockCursor	(Proc)
CalcBounds	(Proc; virtual)
ChangeBounds	(Proc; virtual)
ClearEvent	(Proc)
CommandEnabled	(Function)
Cursor	(Field)
DataSize	(Function; virtual)
DisableCommands	(Proc)
Done	(Destruct; virtual)
DragMode	(Field)
DragView	(Proc)

Draw	(Proc; virtual)
DrawCursor	(Proc; private)
DrawHide	(Proc; private)
DrawShow	(Proc; private)
DrawUnderRect	(Proc; private)
DrawUnderView	(Proc; private)
DrawView	(Proc)
EnableCommands	(Proc)
EndModal	(Proc; virtual)
EventAvail	(Function)
EventMask	(Field)
Execute	(Function; virtual)
Exposed	(Function)
Focus	(Function)
GetBounds	(Proc)
GetClipRect	(Proc)
GetColor	(Function)
GetCommands	(Proc)
GetData	(Proc; virtual)
GetEvent	(Proc; virtual)
GetExtent	(Proc)
GetHelpCtx	(Function; virtual)
GetPalette	(Function; virtual)
GetPeerViewPtr	(Proc)
GetState	(Function)
GrowMode	(Field)
GrowTo	(Proc)
HandleEvent	(Proc; virtual)
HelpCtx	(Field)
Hide	(Proc)
HideCursor	(Proc)
Init	(Construc)
KeyEvent	(Proc)
Load	(Construc)
Locate	(Proc)
MakeFirst	(Proc)
MakeGlobal	(Proc)
MakeLocal	(Proc)
MouseEvent	(Function)
MouseInView	(Function)
MoveTo	(Proc)
Next	(Field)
NextView	(Function)
NormalCursor	(Proc)
Options	(Field)
Origin	(Field)
Owner	(Field)
Prev	(Function)
PrevView	(Function)
PutEvent	(Proc; virtual)
PutInFrontOf	(Proc)
PutPeerViewPtr	(Proc)
ResetCursor	(Proc; virtual; private)

Select	(Proc)
SetBounds	(Proc)
SetCmdState	(Proc)
SetCommands	(Proc)
SetCursor	(Proc)
SetData	(Proc; virtual)
SetState	(Proc; virtual)
Show	(Proc)
ShowCursor	(Proc)
Size	(Field)
SizeLimits	(Proc; virtual)
State	(Field)
Store	(Proc)
TopView	(Function)
Valid	(Function; virtual)
WriteBuf	(Proc)
WriteChar	(Proc)
WriteLine	(Proc)
WriteStr	(Proc)
TGroup	
At	(Function; private)
Awaken	(Proc; virtual)
Buffer	(Field)
ChangeBounds	(Proc; virtual)
Clip	(Field; private)
Current	(Field)
DataSize	(Function; virtual)
Delete	(Proc)
Done	(Destruct; virtual)
Draw	(Proc; virtual)
DrawSubViews	(Proc; private)
EndModal	(Proc; virtual)
EndState	(Field)
EventError	(Proc; virtual)
Execute	(Function; virtual)
ExecView	(Function)
FindNext	(Function; private)
First	(Function)
FirstMatch	(Function; private)
FirstThat	(Function)
FocusNext	(Function)
ForEach	(Proc)
FreeBuffer	(Proc; private)
GetBuffer	(Proc; private)
GetData	(Proc; virtual)
GetHelpCtx	(Function; virtual)
GetSubViewPtr	(Proc)
HandleEvent	(Proc; virtual)
IndexOf	(Function; private)
Init	(Construc)
Insert	(Proc)
InsertBefore	(Proc)

InsertView	(Proc; private)
Last	(Field)
Load	(Construc)
Lock	(Proc)
LockFlag	(Field; private)
Phase	(Field)
phFocused	(Const)
phPostProcess	(Const)
phPreProcess	(Const)
PutSubViewPtr	(Proc)
Redraw	(Proc)
RemoveView	(Proc; private)
ResetCurrent	(Proc; private)
ResetCursor	(Proc; virtual; private)
SelectNext	(Proc)
SetCurrent	(Proc; private)
SetData	(Proc; virtual)
SetState	(Proc; virtual)
Store	(Proc)
Unlock	(Proc)
Valid	(Function; virtual)
TWindow	
Close	(Proc; virtual)
Done	(Destruct; virtual)
Flags	(Field)
Frame	(Field)
GetPalette	(Fnction; virtual)
GetTitle	(Fnction; virtual)
HandleEvent	(Poc; virtual)
Init	(Construc)
InitFrame	(Proc; virtual)
Load	(Construc)
Number	(Field)
Palette	(Field)
SetState	(Proc; virtual)
SizeLimits	(Proc; virtual)
StandardScrollBar	(Function)
Store	(Proc)
Title	(Field)
Zoom	(Proc; virtual)
ZoomRect	(Field)
TDialog	
GetPalette	(Function; virtual)
HandleEvent	(Proc; virtual)
Init	(Construc)
Load	(Construc)
Valid	(Function; virtual)
TFileDialog	
Directory	(Field)
Done	(Destruct; virtual)

	FileList	(Field)
	FileName	(Field)
	GetData	(Proc; virtual)
	GetFileName	(Proc)
	HandleEvent	(Proc; virtual)
	Init	(Construc)
	Load	(Construc)
	ReadDirectory	(Proc; private)
	SetData	(Proc; virtual)
	Store	(Proc)
	Valid	(Function; virtual)
	Wildcard	(Field)
	-TChDirDialog	
	ChDirButton	(Field)
	DataSize	(Function; virtual)
	DirInput	(Field)
	DirList	(Field)
	GetData	(Proc; virtual)
	HandleEvent	(Proc; virtual)
	Init	(Construc)
	Load	(Construc)
	OkButton	(Field)
	SetData	(Proc; virtual)
	SetUpDialog	(Proc; private)
	Store	(Proc)
	Valid	(Function; virtual)
	-TColorDialog	
	BakLabel	(Field)
	BakSel	(Field)
	DataSize	(Function; virtual)
	Display	(Field)
	ForLabel	(Field)
	ForSel	(Field)
	GetData	(Proc; virtual)
	GetIndexes	(Proc)
	GroupIndex	(Field)
	Groups	(Field)
	HandleEvent	(Proc; virtual)
	Init	(Construc)
	Load	(Construc)
	MonoLabel	(Field)
	MonoSel	(Field)
	Pal	(Field)
	SetData	(Proc; virtual)
	SetIndexes	(Proc)
	Store	(Proc)
	-THistoryWindow	
	GetPalette	(Function; virtual)
	GetSlection	(Function; virtual)
	Init	(Construc)

	InitViewer	(Proc; virtual)
	Viewer	(Field)
	TEditWindow	
	Close	(Proc; virtual)
	Editor	(Field)
	GetTitle	(Function; virtual)
	HandleEvent	(Proc; virtual)
	Init	(Construc)
	Load	(Construc)
	SizeLimits	(Proc; virtual)
	Store	(Proc)
	TDesktop	
	Background	(Field)
	Cascade	(Proc)
	HandleEvent	(Proc; virtual)
	Init	(Construc)
	InitBackground	(Proc; virtual)
	Load	(Construc)
	Store	(Proc)
	Tile	(Proc)
	TileColumnsFirst	(Field)
	TileError	(Proc; virtual)
	TProgram	
	CanMoveFocus	(Function)
	Done	(Destruct; virtual)
	ExecuteDialog	(Function)
	GetEvent	(Proc; virtual)
	GetPalette	(Function; virtual)
	HandleEvent	(Proc; virtual)
	Idle	(Proc; virtual)
	Init	(Construc)
	InitDesktop	(Proc; virtual)
	InitMenuBar	(Proc; virtual)
	InitScreen	(Proc; virtual)
	InitStatusLine	(Proc; virtual)
	InsertWindow	(Function)
	OutOfMemory	(Proc; virtual)
	PutEvent	(Proc; virtual)
	Run	(Proc; virtual)
	SetScreenMode	(Proc)
	ValidView	(Function)
	TApplication	
	Cascade	(Proc)
	Done	(Destruct; virtual)
	DosShell	(Proc)
	GetTileRect	(Proc; virtual)
	HandleEvent	(Proc; virtual)
	Init	(Construc)
	Tile	(Proc)
	WriteShellMsg	(Proc; virtual)

—TFrame	
Draw	(Proc; virtual)
FrameLine	(Proc; private)
FrameMode	(Field; private)
GetPalette	(Function; virtual)
HandleEvent	(Proc; virtual)
Init	(Construc)
SetState	(Proc; virtual)
—TScrollBar	
ArStep	(Field)
Chars	(Field; private)
Draw	(Proc; virtual)
DrawPos	(Proc; private)
GetPalette	(Function; virtual)
GetPos	(Function; private)
GetSize	(Function; private)
HandleEvent	(Proc; virtual)
Init	(Construc)
Load	(Construc)
Max	(Field)
Min	(Field)
PgStep	(Field)
ScrollDraw	(Proc; virtual)
ScrollStep	(Function; virtual)
SetParams	(Proc)
SetRange	(Proc)
SetStep	(Proc)
SetValue	(Proc)
Store	(Proc)
Value	(Field)
—TScroller	
ChangeBounds	(Proc; virtual)
CheckDraw	(Proc; private)
Delta	(Field)
DrawFlag	(Field; private)
DrawLock	(Field; private)
GetPalette	(Function; virtual)
HandleEvent	(Proc; virtual)
HScrollBar	(Field)
Init	(Construc)
Limit	(Field)
Load	(Construc)
ScrollDraw	(Proc; virtual)
ScrollTo	(Proc)
SetLimit	(Proc)
SetState	(Proc; virtual)
Store	(Proc)
VScrollBar	(Field)
—TTextDevice	

	StrRead	(Function; virtual)
	StrWrite	(Proc; virtual)
└─	Terminal	
	BufDec	(Proc)
	Buffer	(Field)
	BufInc	(Proc)
	BufSize	(Field)
	CalcWidth	(Function)
	CanInsert	(Function)
	Done	(Destruct; virtual)
	Draw	(Proc; virtual)
	Init	(Construc)
	NextLine	(Function)
	PrevLines	(Function)
	QueBack	(Field)
	QueEmpty	(Function)
	QueFront	(Field)
	StrRead	(Function; virtual)
	StrWrite	(Proc; virtual)
└─	TOutlineViewer	
	Adjust	(Proc; virtual)
	AdjustFocus	(Proc; private)
	CreateGraph	(Function)
	Draw	(Proc; virtual)
	ExpandAll	(Proc)
	FirstThat	(Function)
	Foc	(Field)
	Focused	(Proc; virtual)
	ForEach	(Function)
	GetChild	(Function; virtual)
	GetGraph	(Function; virtual)
	GetNode	(Function)
	GetNumChildren	(Function; virtual)
	GetPalette	(Function; virtual)
	GetRoot	(Function; virtual)
	GetText	(Function; virtual)
	HandleEvent	(Proc; virtual)
	HasChildren	(Function; virtual)
	Init	(Construc)
	IsExpanded	(Function; virtual)
	IsSelected	(Function; virtual)
	Iterate	(Function; private)
	Load	(Construc)
	Selected	(Proc; virtual)
	SetState	(Proc; virtual)
	Store	(Proc)
	Update	(Proc)
└─	TOutline	
	Adjust	(Proc; virtual)
	Done	(Destruct; virtual)

	GetChild	(Function; virtual)
	GetNumChildren	(Function; virtual)
	GetRoot	(Function; virtual)
	GetText	(Function; virtual)
	HasChildren	(Function; virtual)
	Init	(Construc)
	IsExpanded	(Function; virtual)
	Load	(Construc)
	Root	(Field)
	Store	(Proc)
	TListViewer	
	ChangeBounds	(Proc; virtual)
	Draw	(Proc; virtual)
	Focused	(Field)
	FocusItem	(Proc; virtual)
	FocusItemNum	(Proc; virtual; private)
	GetPalette	(Function; virtual)
	GetText	(Function; virtual)
	HandleEvent	(Proc; virtual)
	HScrollBar	(Field)
	Init	(Construc)
	IsSelected	(Function; virtual)
	Load	(Construc)
	NumCols	(Field)
	Range	(Field)
	SelectItem	(Proc; virtual)
	SetRange	(Proc)
	Store	(Proc)
	TopItem	(Field)
	VScrollBar	(Field)
	TListBox	
	DataSize	(Function; virtual)
	GetData	(Proc; virtual)
	GetText	(Function; virtual)
	Init	(Construc)
	List	(Field)
	Load	(Construc)
	NewList	(Proc; virtual)
	SetData	(Proc; virtual)
	Store	(Proc)
	TSortedListBox	
	GetKey	(Function; virtual)
	HandleEvent	(Proc; virtual)
	Init	(Construc)
	NewList	(Proc; virtual)
	SearchPos	(Field)
	ShiftState	(Field)
	TFileList	
	DataSize	(Function; virtual)
	Done	(Destruct; virtual)

	FocusItem	(Proc; virtual)
	GetData	(Proc; virtual)
	GetKey	(Function; virtual)
	GetText	(Function; virtual)
	HandleEvent	(Proc; virtual)
	Init	(Construc)
	ReadDirectory	(Proc)
	SetData	(Proc; virtual)
	TDirListBox	
	Cur	(Field)
	Dir	(Field)
	Done	(Destruct; virtual)
	GetText	(Function; virtual)
	HandleEvent	(Proc; virtual)
	Init	(Construc)
	IsSelected	(Function; virtual)
	NewDirectory	(Proc)
	SetState	(Proc; virtual)
	THistoryViewer	
	GetPalette	(Function; virtual)
	GetText	(Function; virtual)
	HandleEvent	(Proc; virtual)
	HistoryId	(Field)
	HistoryWidth	(Function)
	Init	(Construc)
	TColorGroupList	
	Done	(Destruct; virtual)
	FocusItem	(Proc; virtual)
	GetGroup	(Function)
	GetGroupIndex	(Function)
	GetNumGroups	(Function)
	GetText	(Function; virtual)
	Groups	(Field)
	HandleEvent	(Proc; virtual)
	Init	(Construc)
	Load	(Construc)
	SetGroupIndex	(Proc)
	Store	(Proc)
	TColorItemList	
	FocusItem	(Proc; virtual)
	GetText	(Function; virtual)
	HandleEvent	(Proc; virtual)
	Init	(Construc)
	Items	(Field)
	TMenuView	
	Current	(Field)
	Execute	(Function; virtual)
	FindItem	(Function)
	GetHelpCtx	(Function; virtual)

	GetItemRect	(Proc; virtual)
	GetPalette	(Function; virtual)
	HandleEvent	(Proc; virtual)
	HotKey	(Function)
	Init	(Construc)
	Load	(Construc)
	Menu	(Field)
	NewSubView	(Function; virtual)
	ParentMenu	(Field)
	Store	(Proc)
—	TMenuBar	
	Done	(Destruct; virtual)
	Draw	(Proc; virtual)
	GetItemRect	(Proc; virtual)
	Init	(Construc)
—	TMenuBox	
	Draw	(Proc; virtual)
	GetItemRect	(Proc; virtual)
	Init	(Construc)
—	TMenuPopup	
	HandleEvent	(Proc; virtual)
	Init	(Construc)
—	TStatusLine	
	Defs	(Field)
	Done	(Destruct; virtual)
	Draw	(Proc; virtual)
	DrawSelect	(Proc; private)
	FindItems	(Proc; private)
	GetPalette	(Function; virtual)
	HandleEvent	(Proc; virtual)
	Hint	(Function; virtual)
	Init	(Construc)
	Items	(Field)
	Load	(Construc)
	Store	(Proc)
	Update	(Proc; virtual)
—	TInputLine	
	CanScroll	(Function; private)
	CurPos	(Field)
	Data	(Field)
	DataSize	(Function; virtual)
	Done	(Destruct; virtual)
	Draw	(Proc; virtual)
	FirstPos	(Field)
	GetData	(Proc; virtual)
	GetPalette	(Function; virtual)
	HandleEvent	(Proc; virtual)
	Init	(Construc)
	Load	(Construc)

MaxLen	(Field)
SelectAll	(Proc)
SelEnd	(Field)
SelStart	(Field)
SetData	(Proc; virtual)
SetState	(Proc; virtual)
SetValidator	(Proc)
Store	(Proc)
Valid	(Function; virtual)
Validator	(Field)
TFileInputLine	
HandleEvent	Proc; virtual)
Init	(Construc)
TButton	
AmDefault	(Field)
Command	(Field)
Done	(Destruct; virtual)
Draw	(Proc; virtual)
DrawState	(Proc)
Flags	(Field)
GetPalette	(Function; virtual)
HandleEvent	(Proc; virtual)
Init	(Construc)
Load	(Construc)
MakeDefault	(Proc)
Press	(Proc; virtual)
SetState	(Proc; virtual)
Store	(Proc)
Title	(Field)
TCluster	
ButtonState	(Function)
Column	(Function; private)
DataSize	(Function; virtual)
Done	(Destruct; virtual)
DrawBox	(Proc)
DrawMultiBox	(Proc)
EnableMask	(Field)
FindSel	(Function; private)
GetData	(Proc; virtual)
GetHelpCtx	(Function; virtual)
GetPalette	(Function; virtual)
HandleEvent	(Proc; virtual)
Init	(Construc)
Load	(Construc)
Mark	(Function; virtual)
MultiBox	(Proc; virtual)
MultiBox	(Function; virtual)
Press	(Proc; virtual)
Row	(Function; private)
Sel	(Field)

	SetButtonState	(Proc)
	SetData	(Proc; virtual)
	SetState	(Proc; virtual)
	Store	(Proc)
	Strings	(Field)
	Value	(Field)
—	TRadioButtons	
	Draw	(Proc; virtual)
	Mark	(Function; virtual)
	MovedTo	(Proc; virtual)
	Press	(Proc; virtual)
	SetData	(Proc; virtual)
—	TCheckBoxes	
	Draw	(Proc; virtual)
	Mark	(Function; virtual)
	Press	(Proc; virtual)
—	TMultiCheckBoxes	
	DataSize	(Function; virtual)
	Done	(Destruct; virtual)
	Draw	(Proc; virtual)
	Flags	(Field)
	GetData	(Proc; virtual)
	Init	(Construc)
	Load	(Construc)
	MultiMark	(Function; virtual)
	Press	(Proc; virtual)
	SelRange	(Field)
	SetData	(Proc; virtual)
	States	(Field)
	Store	(Proc)
—	TMonoSelector	
	Draw	(Proc; virtual)
	HandleEvent	(Proc; virtual)
	Init	(Construc)
	Mark	(Function; virtual)
	MovedTo	(Proc; virtual)
	NewColor	(Proc)
	Press	(Proc; virtual)
—	TStaticText	
	Done	(Destruct; virtual)
	Draw	(Proc; virtual)
	GetPalette	(Function; virtual)
	GetText	(Proc; virtual)
	Init	(Construc)
	Load	(Construc)
	Store	(Proc)
	Text	(Field)
—	TParamText	

	DataSize	(Function; virtual)
	GetText	(Proc; virtual)
	Init	(Construc)
	Load	(Construc)
	ParamCount	(Field)
	ParamList	(Field)
	SetData	(Proc; virtual)
	Store	(Proc)
	TLabel	
	Draw	(Proc; virtual)
	GetPalette	(Function; virtual)
	HandleEvent	(Proc; virtual)
	Init	(Construc)
	Light	(Field)
	Link	(Field)
	Load	(Construc)
	Store	(Proc)
	THistory	
	Draw	(Proc; virtual)
	GetPalette	(Function; virtual)
	HandleEvent	(Proc; virtual)
	HistoryId	(Field)
	Init	(Construc)
	InitHistoryWindow	(Function; virtual)
	Link	(Field)
	Load	(Construc)
	RecordHistory	(Proc; virtual)
	Store	(Proc)
	TBackground	
	Draw	(Proc; virtual)
	GetPalette	(Function; virtual)
	Init	(Construc)
	Load	(Construc)
	Pattern	(Field)
	Store	(Proc)
	TFileInfoPane	
	Draw	(Proc; virtual)
	GetPalette	(Function; virtual)
	HandleEvent	(Proc; virtual)
	Init	(Construc)
	S	(Field)
	TColorSelector	
	Color	(Field)
	Draw	(Proc; virtual)
	HandleEvent	(Proc; virtual)
	Init	(Construc)
	Load	(Construc)
	SetType	(Field)
	Store	(Proc)

└─TColorDisplay	
Color	(Field)
Done	(Destruct; virtual)
Draw	(Proc; virtual)
HandleEvent	(Proc; virtual)
Init	(Construc)
Load	(Construc)
SetColor	(Proc; virtual)
Store	(Proc)
Text	(Field)
└─TIndicator	
Draw	(Proc; virtual)
GetPalette	(Function; virtual)
Init	(Construc)
Location	(Field)
Modified	(Field)
SetState	(Proc; virtual)
SetValue	(Proc)
└─TEditor	
AutoIndent	(Field)
BufChar	(Function)
Buffer	(Field)
BufLen	(Field)
BufPtr	(Function)
BufSize	(Field)
CanUndo	(Field)
ChangeBounds	(Proc; virtual)
CharPos	(Function; private)
CharPtr	(Function; private)
ClipCopy	(Function; private)
ClipCut	(Proc; private)
ClipPaste	(Proc; private)
ConvertEvent	(Proc; virtual)
CurPos	(Field)
CurPtr	(Field)
CursorVisible	(Function)
DelCount	(Field)
DeleteRange	(Proc; private)
DeleteSelect	(Proc)
Delta	(Field)
Done	(Destruct; virtual)
DoneBuffer	(Proc; virtual)
DoSearchReplace	(Proc; private)
DoUpdate	(Proc; private)
Draw	(Proc; virtual)
DrawLine	(Field)
DrawLines	(Proc; private)
DrawPtr	(Field)
Find	(Proc; private)
FormatLine	(Proc; private)

GapLen	(Field)
GetMousePtr	(Function; private)
GetPalette	(Function; virtual)
HandleEvent	(Proc; virtual)
HasSelection	(Function; private)
HideSelect	(Proc; private)
HScrollBar	(Field)
Indicator	(Field)
Init	(Construc)
InitBuffer	(Proc; virtual)
InsCount	(Field)
InsertBuffer	(Function)
InsertFrom	(Function; virtual)
InsertText	(Function)
IsClipboard	(Function; private)
IsValid	(Field)
KeyState	(Field; private)
Limit	(Field)
LineEnd	(Function; private)
LineMove	(Function; private)
LineStart	(Function; private)
Load	(Construc)
Lock	(Proc; private)
LockCount	(Field; private)
Modified	(Field)
NewLine	(Proc; private)
NextChar	(Function; private)
NextLine	(Function; private)
NextWord	(Function; private)
Overwrite	(Field)
PrevChar	(Function; private)
PrevLine	(Function; private)
PrevWord	(Function; private)
Replace	(Proc; private)
ScrollTo	(Proc)
Search	(Function)
Selecting	(Field)
SelEnd	(Field)
SelStart	(Field)
SetBufLen	(Proc; private)
SetBufSize	(Function; virtual)
SetCmdState	(Proc)
SetCurPtr	(Proc; private)
SetSelect	(Proc)
SetState	(Proc; virtual)
StartSelect	(Proc; private)
Store	(Proc)
ToggleInsMode	(Proc; private)
TrackCursor	(Proc)
Undo	(Proc)
Unlock	(Proc; private)
Update	(Proc; private)
UpdateCommands	(Proc; virtual)

UpdateFlags	(Field; private)
Valid	(Function; virtual)
VScrollBar	(Field)
TMemo	
DataSize	(Function; virtual)
GetData	(Proc; virtual)
GetPalette	(Function; virtual)
HandleEvent	(Proc; virtual)
Load	(Construc)
SetData	(Proc; virtual)
Store	(Proc)
TFileEditor	
DoneBuffer	(Proc; virtual)
FileName	(Field)
HandleEvent	(Proc; virtual)
Init	(Construc)
InitBuffer	(Proc; virtual)
Load	(Construc)
LoadFile	(Function)
Save	(Function)
SaveAs	(Function)
SaveFile	(Function)
SetBufSize	(Function; virtual)
Store	(Proc)
UpdateCommands	(Proc; virtual)
Valid	(Function; virtual)
TValidator	
Error	(Proc; virtual)
Init	(Construc)
IsValid	(Function; virtual)
IsValidInput	(Function; virtual)
Load	(Construc)
Options	(Field)
Status	(Field)
Store	(Proc)
Transfer	(Function; virtual)
Valid	(Function)
TPXPictureValidator	
Done	(Destruct; virtual)
Error	(Proc; virtual)
Init	(Construc)
IsValid	(Function; virtual)
IsValidInput	(Function; virtual)
Load	(Construc)
Pic	(Field)
Picture	(Function; virtual)
Store	(Proc)
TFilterValidator	
Error	(Proc; virtual)

Init	(Construc)
IsValid	(Function; virtual)
IsValidInput	(Function; virtual)
Load	(Construc)
Store	(Proc)
ValidChars	(Field)
TRangeValidator	
Error	(Proc; virtual)
Init	(Construc)
IsValid	(Function; virtual)
Load	(Construc)
Max	(Field)
Min	(Field)
Store	(Proc)
Transfer	(Function; virtual)
TLookupValidator	
IsValid	(Function; virtual)
Lookup	(Function; virtual)
TStringLookupValidator	
Done	(Destruct; virtual)
Error	(Proc; virtual)
Init	(Construc)
Load	(Construc)
Lookup	(Function; virtual)
NewStringList	(Proc)
Store	(Proc)
Strings	(Field)
-T Point	
X	(Field)
Y	(Field)
- T Rect	
A	(Field)
A ssign	(Proc)
B	(Field)
Contains	(Function)
Copy	(Proc)
Empty	(Function)
Equals	(Function)
Grow	(Proc)
Intersect	(Proc)
Move	(Proc)
Union	(Proc)

Содержание

От автора	3
ГЛАВА 1. Turbo Vision. Ключевые понятия	5
Управление данными или управление событиями?	5
Определение новых событий	7
Метод Idle	8
Функция Message	9
Отображаемые и неотображаемые объекты	10
Отображаемые объекты	10
Координатная система и объект TRect	11
Модальные и выбранные объекты	12
Групповые объекты	13
Неотображаемые объекты	14
Разное	14
Функция CStrLen	14
Константа ErrorAttr	15
Процедура FormatStr	15
Функция NewSItem	16
Заключение	17
ГЛАВА 2. Базовые объекты: TApplication и TProgram.	18
TApplication: объект-приложение	18
Инициализация	19
Выполнение	20
Завершение	21
Цветовая палитра приложения	21
Изменения в Turbo Vision 2.0(A)	23
Метод Cascade	23
Метод DosShell	23
Метод GetTileRect	24
Метод HandleEvent	24
Метод Tile	24
Метод WriteShellMsg	25
Часы в приложении	25

TProgram: объект-программа	27
Метод GetEvent	27
Метод HandleEvent	28
Метод Idle	28
OutOfMemory	28
Метод PutEvent	29
Изменения в Turbo Vision 2.0(Б)	29
Метод CanMoveFocus	29
Метод ExecuteDialog	29
Метод InsertWindow	29
TDesktop: Рабочая область	30
Изменения в Turbo Vision 2.0.Объект TDesktop	32
Поле TileColumnsFirst	32
Методы Load и Store	34
TStatusLine: Строка состояния	34
Отображение сообщений в строке состояния	37
TMenuView: Полоса меню	39
Горизонтальное меню	39
Доступ к структуре меню	39
Объект TMenuPopUp	42
Как поместить элемент меню в самую левую позицию	44
Как сделать меню с маркерами	44
Расширения	49
Использование справочного контекста	49
О сохранении экрана	51
Программы без меню	55
Скрытая строка состояния	56
Режим 132x25	57
Динамически изменяемые меню и строки состояния	59
Динамически изменяемое меню	60
Строка состояния	62
Модуль APP	64
Стандартные меню и строки состояния	64
Заключение	67
 ГЛАВА 3. Окна и панели диалога.	 68
Окна	68
Установка свойств окна	71
Отображение информации в окне	71

Модальные окна	74
Меню в окне	76
Использование объекта TFrame	79
Панели диалога	81
Стандартные панели диалога	82
Изменения в Turbo Vision 2.0	82
Функция InputBox	83
Функция InputBoxRect	84
Объект TFileDialog	84
Расширение функциональности	86
Элементы управления	88
Элементы управления и фокус	89
Объект TButton	90
Нестандартные кнопки	92
Объект TCluster	96
Кнопки с зависимой и независимой фиксацией	96
Изменения в Turbo Vision 2.0. Объект TCluster	99
Поле EnableMask	99
Метод ButtonState	100
Метод DrawMultiBox	100
Метод MultiMark	100
Метод SetButtonState	100
Объект TListBox	100
Список с возможностью выбора нескольких элементов	105
Объект TStaticText	110
Объект TLabel	112
Объект TParamText	114
Объект THistory	115
Использование протокола	121
Изменения в Turbo Vision 2.0. Объект THistory	122
Метод RecordHistory	122
Объект TInputLine	122
Обмен данными между строками ввода	124
Изменения в Turbo Vision 2.0. Объект TInputLine	126
Поле Validator	126
Метод SetValidator	126
Расширения строки ввода	126
Объект TInputLineUC	127
Объект TInputLineLC	128
Объект TInputLinePS	128

Объект TFilterInput	131
Объект TInputLineNE	133
Объект TScrollBar	133
Использование полос прокрутки	137
Перемещаемые элементы управления	139
Отображение статуса операции	143
Отображение разноцветного текста	145
Панель диалога с возможностью прокрутки информации	146
Модуль OUTLINE	150
Модуль DIALOGS	151
Модуль STDDLГ	152
Модуль VIEWS	152
Заключение	154
ГЛАВА 4. Редактор и средства просмотра текста.	155
Объект TEditor	155
Использование области обмена данными	157
Объект TMemo	160
Объект TFileEditor	162
Объект TTerminal	163
Отображение текста в 16-ричном виде	164
Модуль EDITORS	166
ГЛАВА 5. Цветовые палитры.	168
Устройство палитр	168
Изменение цветов	170
Настройка цветов	174
Индексы стандартной палитры	176
Загрузка палитр	178
Модуль COLORSEL	179
Заключение	180
ГЛАВА 6. Модуль VALIDATE. Объекты проверки ввода.	181
Объект TValidator	183

Метод Error	183
Метод IsValid	183
Метод IsValidInput	184
Метод Valid	184
Объект TFilterValidator	184
Метод IsValidInput	184
Объект TPXPictureValidator	184
Метод IsValidInput	185
Метод Picture	185
Объект TRangeValidator	186
Объект TLookupValidator	186
Объект TStringLookupValidator	186
Заключение	189
 ГЛАВА 7. Использование справочной системы.	 190
Некоторые замечания по расширению справочной системы	195
Заключение	196
 ГЛАВА 8. Отладка Turbo Vision-приложений.	 198
Модуль TVDEBUG	201
Средства, входящие в Turbo Vision Development Kit	202
 ГЛАВА 9. Неотображаемые объекты.	 205
Коллекции	205
Как создаются и используются коллекции объектов	205
Отсортированные и неотсортированные коллекции	206
Как коллекции используются отображаемыми объектами	208
Коллекции и требования к памяти	209
Потоки	209
Методы Load и Store	211
Изменения в Turbo Vision 2.0	212
Переменная StreamError	212
Объект TMemoryStream	214
Пути расширения потоков	214
Ресурсы	215

Создание файла ресурсов	215
Использование файла ресурсов	216
Метод TResourceFile.SwitchTo	219
Доступ к ресурсам в файлах	219
Объекты TStringList и TStrListMaker	221
Заключение	223

ПРИЛОЖЕНИЕ А 224

Регистрационные коды	224
Реакция стандартных объектов на события	225
Объект TApplication	225
Объект TButton	225
Объект TCluster	225
Объект TColorDialog	226
Объект TColorDisplay	226
Объект TColorGroupList	226
Объект TColorItemList	226
Объект TColorSelector	226
Объект TDesktop	227
Объект TDialog	227
Объект TEditor	227
Объект TEditWindow	227
Объект TGroup	228
Объект TProgram	228
Объект TInputLine	228
Объект TLabel	228
Объект TMonoSelector	229
Объект THistoryViewer	229
Объект THistory	229
Объект TMemo	229
Объект TMenuPopup	230
Объект TMenuView	230
Объект TFileEditor	230
Объект TStatusLine	230
Объект TOutlineViewer	231
Объект TFileInputLine	231
Объект TSortedListBox	231
Объект TFileList	231
Объект TFileInfoPane	232
Объект TFileDialog	232

Объект TDirListBox	232
Объект TChDirDialog	232
Объект TView	232
Объект TFrame	233
Объект TScrollBar	233
Объект TScroller	233
Объект TListViewer	234
Объект TWindow	234

ПРИЛОЖЕНИЕ Б 235

TURBO VISION 2.0 .Расширенная иерархия объектов	235
---	-----

Издательство "ДИАЛЕКТИКА" **предлагает Вашему вниманию:**

Гради Буч, **"Объектно-ориентированное проектирование с примерами применения"**, 528 стр., твердый переплет, книга вышла в свет.

"ObjectWindows для C++", в 2 томах по 280 стр., мягкий переплет, слайд, книга вышла в свет.

"Turbo Vision для языка Паскаль", 2 тома по 320 стр., мягкий переплет, слайд, книга вышла в свет.

"Справочное руководство по FoxPro 2.0", в 3 томах по 620 стр., твердый переплет, книга вышла в свет.

А.Федоров, Д.Рогаткин **"Borland Pascal в среде Windows"**, 512 стр, твердый переплет. Ориентировочный выход в свет - сентябрь 1993 г.

"Справочник по процедурам и функциям Borland Pascal 7.0", 260 стр., мягкий переплет. Ориентировочный выход в свет- октябрь-ноябрь 1993 г.

"Использование Turbo Assembler при разработке программ", 288 стр., мягкий переплет. Ориентировочный выход в свет- октябрь-ноябрь 1993 г.

"Особенности программирования на Borland Pascal", 260 стр., мягкий переплет. Ориентировочный выход в свет - ноябрь-декабрь 1993 г.

"Справочник по функциям Borland C++ 3.1", 300 стр., мягкий переплет. Ориентировочный выход в свет - декабрь 1993 г.

С.Шлеер и С.Меллор, **"Объектно-ориентированный анализ: моделирование мира в состояниях"**, 300 стр., мягкая обложка. Ориентировочный выход в свет - декабрь 1993 г.

Книга Гради Буча **"Объектно-ориентированное проектирование с примерами применения"** - 528 стр., твердый переплет.

Аннотация:

Предлагаемая Вашему вниманию книга "Object Oriented Design with Application" by Grady Booch" (в издании на русском языке - "Объектно-ориентированное проектирование с примерами применения") вышла в издательстве The Benjamin/Cummings Publishing, Inc. (США) в 1991 году и является одной из самых продаваемых.

Об авторе. Гради Буч - основатель и директор американской фирмы Object-Oriented Products at Rational, образованной в 1980 году. Был одним из первых в США, кто начал использование объектно-ориентированного метода проектирования с применением различных объектных и объектно-ориентированных языков программирования. Г-н Буч является членом редколлегии американских журналов: Object Magazin, Journal of Object - Oriented Programming, HotLine of Object - Oriented Technology.

О книге. "Объектно-ориентированное проектирование с примерами применения" представляет собой первое полное изложение объектно-ориентированной методологии и ее компонент. Книга разделена автором на три основные части.

Часть первая - КОНЦЕПЦИИ. В этой части обсуждаются свойства сложных систем, рассматривается понятие декомпозиции, какую роль при этом играют абстракции, иерархия. Понятие конструирования системы определено как процесс создания модели реального объекта. Автором рассмотрены фундаментальные основы объектного подхода и произведен анализ развития языков программирования. В качестве элементов объектного подхода выделяются процессы объектно-ориентированного анализа (OOA), проектирования (OOD) и программирования (OOP). Приведены направления где практические результаты использования этого подхода уже получены. Автор подчеркивает значения Классов и Объектов в процессе проектирования и дает рекомендации по оценке качества этих абстракций. Содержатся примеры структур классов и объектов, оптимизирующих программные системы. Отдельная глава посвящена выработке правил классификации классов и объектов, рассматриваются три основных типа классификаций:

- Классическое распределение по категориям;
- Концептуальное группирование;
- Теория прототипирования.

Часть вторая - МЕТОДОЛОГИЯ. Здесь рассматривается система обозначений, позволяющая описать проект не прибегая к средствам языка реализации проекта. Автором описан собственно процесс объектно-ориентированного проектирования. выделены основные этапы этого процесса, рассмотрены содержание и результаты каждого из них. В этой части также рассмотрены практические аспекты конструирования реальных программных систем, а также основные инструменты, облегчающие труд программиста. Даны некоторые советы, как перейти к объектно-ориентированному проектированию.

Часть третья - ПРИМЕРЫ ПРИМЕНЕНИЯ. В этой части приводятся примеры применения на следующих языках программирования: Smalltalk, Object Pascal, C++, Common Lisp Object System, Ada.

Особое достоинство книги - обширная библиография на сорока страницах, в которой представлены публикации по всем аспектам технологии объектно-ориентированного программирования.

Книга рассчитана на профессиональных программистов, руководителей больших программных проектов и студентов, будущая профессиональная деятельность которых связана с разработкой сложных программных систем.

Книга С.Шлеер и С.Меллора **"Объектно-ориентированный анализ: моделирование мира в состояниях"** - 250 стр., мягкая обложка.

Аннотация:

Объектно-ориентированная методология (ООМ) и ее компоненты: объектно-ориентированный анализ (ООА), объектно-ориентированное проектирование (ООД) и объектно-ориентированное программирование (ООР), стали в настоящее время одной из самых популярных тем в области информатики. Известно, что благодаря объектно-ориентированной методологии, решение проблем разработки

сложного программного обеспечения существенно упрощается. Однако, вопросам реализации или, собственно, языкам программирования придается сегодня преувеличенное значение в ущерб вопросам высокоуровневого проектирования программных систем.

Предлагаемая Вашему вниманию книга известных американских специалистов С.Шлеер и С.Меллора посвящена изложению самых первых этапов процесса разработки сложных (программных, технических или других) систем. В ней детально описывается один из наиболее нетривиальных элементов объектного подхода - процесс объектно-ориентированного анализа. Методология ООА разработана в книге как метод для отождествления важных сущностей в задачах реального мира, для объяснения и понимания того, как они взаимодействуют между собой.

Изложенное представление ООА целенаправленно адресовано программистам-профессионалам и всем, кто занимается программированием.

Данный метод анализа изложен в виде примеров и руководящих принципов, которые могут быть усвоены довольно быстро.

После разбиения системы программного обеспечения на ряд четко определенных предметных областей каждая из них анализируется независимо от остальных в три этапа. В процессе информационного моделирования выделяются концептуальные сущности (объекты), которые составляют подсистему для анализа. С помощью моделей состояний исследуется поведение идентифицированных объектов и связей между ними во времени. Каждое действие, происходящее в системе, рассматривается на этапе моделирования процессов. Созданные модели в дальнейшем могут быть непосредственно преобразованы в объектно-ориентированный проект с помощью рекурсивного проектирования (RD), описанного в книге. Авторы также приводят не зависимую от языка графическую нотацию для описания проектирования программы, библиотеки или среды. Завершает книгу краткий англо-русский словарь терминов наиболее часто употребляемых при объектно-ориентированном подходе.

Книга **"Turbo Vision для языка Паскаль"**, 2 тома по 320 стр., мягкий переплет, слайд.

Аннотация:

Книга описывает объектно-ориентированную библиотеку - программный продукт нового поколения, в котором впервые на базе процедурного языка реализована концепция событийного программирования. В этой книге наглядно демонстрируется то, как следует проектировать и реализовывать систему взаимодействующих объектов, а также показан механизм записи функционирующей системы объектов на диск. Описываемое инструментальное средство предназначено для проектирования интерактивной интерфейсной оболочки пользователя в текстовом режиме, в соответствии со стандартом SAA/CUI.

Книга содержит полную справочную информацию о всех стандартных объектах и методах объектно-ориентированной библиотеки Turbo Vision.

Для программистов разной квалификации, для всех использующих язык программирования Turbo Pascal.

Книга **"Использование Turbo Assembler при разработке программ"**, -288 стр., мягкий переплет.

Аннотация:

В книге описывается одна из наиболее распространенных систем программирования на языке ассемблер для персональных компьютеров - Turbo Assembler 3.x (TASM). Приводится большое число примеров программ и обширная справочная информация по системе TASM. Книга состоит из восьми частей. Материал излагается в порядке возрастания сложности, поэтому тем, кто только начинает изучать ассемблер, могут переходить сразу к изучению нужных им тем.

Часть первая содержит первоначальные сведения об ассемблере, описание архитектуры процессора 8086, основы организации загрузки и выполнения программ в DOS, а также подробное описание инструкций процессора 8086.

Часть вторая посвящена применению ассемблера для разработки программ в реальном режиме.

Часть третья описывает архитектуру процессора 80386, инструкции защищенного режима и возможности Турбо

Ассемблера по написанию программ, использующих защищенный режим.

Часть четвертая подробно излагает основные понятия технологии объектно-ориентированного программирования, содержит описание директив Турбо Ассемблера и примеры программ, реализующих эту технологию.

Часть пятая предназначена для программистов, работающих на языках высокого уровня, и описывает интерфейсы между Турбо Ассемблером и компиляторами с языков C++, Pascal, Basic и Prolog, разработанными фирмой Borland Int.

Часть шестая излагает основные концепции программирования в среде Microsoft Windows, содержит примеры программ для этой среды.

Часть седьмая посвящена описанию архитектуры и инструкций математических сопроцессоров x87, содержит примеры программ, использующих сопроцессор.

Часть восьмая носит справочный характер, содержит описание основных прерываний DOS и Windows, краткий перечень всех процессоров x86 и сопроцессоров x86, описание форматов исполняемых файлов для DOS и Windows, перечень сообщений, генерируемых Турбо Ассемблером в случае обнаружения ошибки в тексте программы, и другие сведения по практическому применению Турбо ассемблера для разработки программ.

В книге **"Особенности программирования на Borland Pascal"** содержится описание новых для Turbo Pascal разделов программирования в системе Borland Pascal with Objects 7.0. Рассматриваются новые операторы, добавленные к языку Паскаль в версии 7.0, новые возможности языка, такие как: открытые параметры в процедурах и функциях (открытые строки и открытые массивы), а также работа со строками, заканчивающимися нулем.

Примерно 80% книги составляет рассмотрение такого принципиально нового вопроса как программирование для защищенного режима процессора. Эти сведения будут

полезными как для программирующих для Windows, так программистов, работающих с DOS в защищенном режиме.

Детально описываются аппаратные возможности процессоров 80286, 80386 и 80486 (с точки зрения программиста).

Описывается интерфейс DOS с защищенным режимом (DPMI) и тонкости программирования в защищенном режиме.

Эти сведения будут полезными также программирующим на С и на ассемблере. Также детально описана технология создания динамически-загружаемых библиотек: для Windows, для защищенного режима DOS и универсальных библиотек. Описаны тонкости написания DLL, типичные проблемы и пути их решения.

Книга содержит также ряд полезных приложений:

- формат выполняемого файла Windows и менеджера времени выполнения NEW EXE.
- список функций DPMI V0.9. и многое другое.

Книга рассчитана на программистов, знакомых с предыдущими версиями Turbo Pascal и позволит максимально использовать возможности нового компилятора.

P.S. Известный американский программист Эндрю Шульман в одной из своих статей сказал:

"Идея фирмы Microsoft о том, что хорошая программа должна обладать способностью работать в реальном режиме относится к разряду недоразумений".

Итак, господа, в скором времени программирование для реального режима работы микропроцессора уйдет в небытие.

Чтобы не отстать от жизни читайте книгу "Особенности программирования на Borland Pascal".

Книга **Дм. Рогаткина и А. Федорова "Borland Pascal в среде Windows"** - 512 стр., твердый переплет.

Аннотация:

Поздравляем Вас с решением заниматься программированием под Windows. Так как этот продукт в 1990-92 гг. сделал много шума, то можно считать, что через несколько лет почти каждая программа будет работать под Windows.

Почти 10 лет назад фирма Borland выбросила на рынок свой легендарный компилятор Turbo Pascal. В то время память компьютеров считалась еще в килобайтах и тактовая частота не превышала 10 МГц. Уже тогда Turbo Pascal обгонял свое время, обладая необычайным комфортом и очень высокой скоростью компиляции, поэтому название Turbo Pascal происходит от слова Turbo, что означает быстрый. Появление на рынке программных средств очередной версии компилятора Borland Pascal говорит о том, что это направление является приоритетным для фирмы Borland.

Книга посвящена основам программирования в среде Microsoft Windows на языке Pascal, но многие общие аспекты программирования, особенно обсуждение интерфейса Windows API, также могут быть полезны пользователям компиляторов Borland C++, Turbo C++ for Windows и Microsoft C совместно с Microsoft Windows SDK.

Что Вас ждет в этой книге ?

Авторы детально описывают процесс создания Windows-программы на основе функций Windows API и объектно-ориентированной библиотеки ObjectWindows.

Книга состоит из следующих основных разделов: компилятор Turbo Pascal 1.5; программная организация Windows; введение в элементы интерфейса Windows; создание простейшей программы для Windows; сообщения; введение в библиотеку Object Windows; окно; меню; панель диалога; элементы управления; ввод данных, клавиатура; работа с манипулятором "мышь"; введение в GDI Windows; курсоры, иконки и растровые изображения; работа с текстом; использование принтера; работа с файловой системой; динамические библиотеки; управление памятью; средства обмена данными; краткий обзор Borland Pascal with Objects 7.0. В книге Дм. Рогаткина и А. Федорова "Borland Pascal в среде Windows" в очень компактной и сжатой форме (без ущерба для полноты) собрана обширная информация по технологии разработки приложений для Windows, что может сделать ее незаменимой при практической разработке программ.

Справочное руководство по FoxPro 2.0 в трех томах, по 520 стр., твердый переплет.

Аннотация :

Данное руководство предлагает читателю полное описание одной из самых лучших систем управления базами данных для персональных компьютеров FoxPro 2.0. Трехтомное издание хорошо иллюстрировано, содержит много программ, написанных в системе FoxPro, богато примерами, показывающими ее лучшие качества и возможности. Предназначено для широкого круга специалистов по вычислительной технике и базам данных.

Справочник по функциям Borland C++ 3.1, 300 с., мягкий переплет.

Аннотация:

Книга содержит полное и подробное описание всех функций, глобальных переменных, констант и типов, описанных в поставляемых с компилятором библиотеках.

Описание каждой функции содержит ее синтаксис, ссылку на библиотеку, в которой описана, детальную информацию о входных параметрах и возвращаемом значении, ссылки на другие функции, связанные с ней логически. Указаны особенности использования функций, характерные ошибки и возможности их устранения.

Функции упорядочены по алфавиту, но дополнительно имеется перечень функций по роду выполняемой операции, а также по заголовочным файлам, в котором они определяются. Справочник разрабатывался так, чтобы его было удобно использовать при работе в среде Borland C++ как замена и дополнение встроенной подсказки.

Рассчитан на программистов, использующих в своей работе Borland C++ 3.x, а также всех, кто интересуется или изучает C++.

Фирма "Диалектика" предоставляет возможность всем желающим получить издаваемую литературу по почте. Для этого Вам необходимо выслать заявку с указанием названия книги и количества заказываемых экземпляров по адресу: *Украина, 252124 г.Киев-124 а/я 506, "Диалектика"*. После выхода в свет Вы окажетесь ее первым обладателем, потому что при ограниченности тиража предпочтение отдается тем, кто обратился непосредственно к фирме. Вы также сможете избежать наценки торгующих организаций, которые в удаленных регионах многократно превышают стоимость самой книги. Приславшие заявки вносятся в базу данных, и периодически получают тематический план с аннотациями готовящихся к выпуску изданий.



Обращаем Ваше внимание на то, что при получении книги по почте мы гарантируем возврат денег (за вычетом почтовых расходов), если книга не удовлетворяет Вашим требованиям.

Рассылку литературы осуществляют

для жителей Украины - фирма "Диалектика" тел. (044) 266-4074

для жителей России - еженедельник "СофтМаркет" тел. (095) 903-2147

Киевлян и гостей города мы приглашаем в магазин "Наука и техника", где расположен консультационный центр фирмы. В нем Вы сможете приобрести книги по самым современным технологиям программирования, заказать техническую литературу американских и западноевропейских издательств за наличный и безналичный расчет.

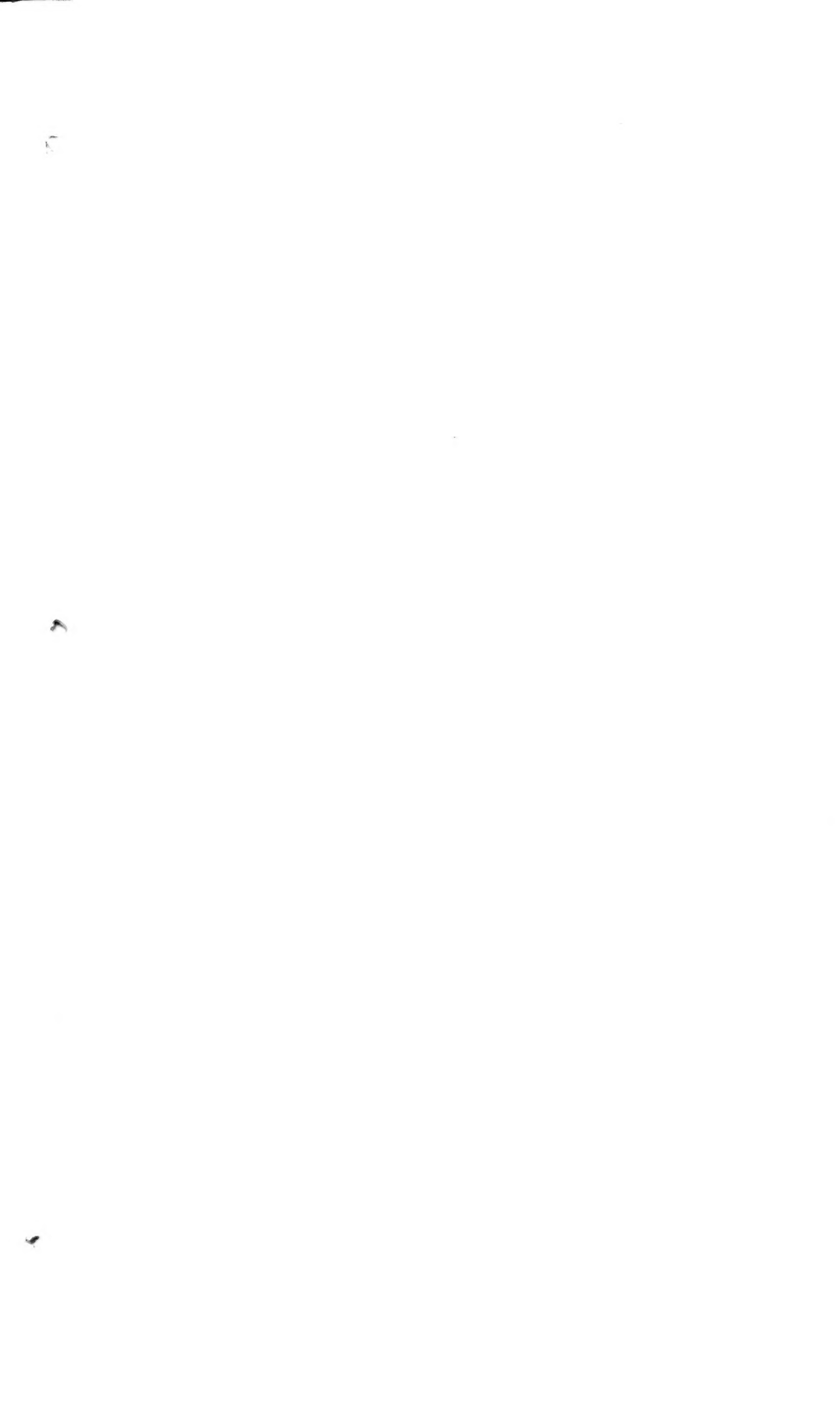
Магазин "Наука и техника"

тел. 559-63-63

г.Киев ул.Строителей 4.

Ехать: М."Дарница" автобус 10, 45, 50. М."Черниговская"

Трамвай 21, 27, 33 до Ленинградской площади





Издательство "Диалектика"

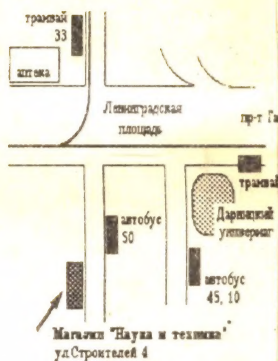
Украина 252124, Киев-124, а/я 506
телефон/факс (044) 266-4074

Приглашаем Вас в магазин "Наука и техника", где расположен консультационный центр фирмы. В нем Вы сможете приобрести книги по самым современным компьютерным технологиям без торгашенки, а также заказать компьютерную литературу американских западноевропейских издательств наличный и безналичный расчет. В центре организована подписка на готовящиеся выпуски издания.

Магазин "Наука и техника"

ул. Строителей 4,
тел. (044) 559-63-63

Ехать: М. "Дарница"
автобус № 10, 45, 50
М. "Черниговская"
трамвай № 21, 27, 33
до Ленинградской
площади



MaxLen	(Field)
SelectAll	(Proc)
SelEnd	(Field)
SelStart	(Field)
SetData	(Proc; virtual)
SetState	(Proc; virtual)
SetValidator	(Proc)
Store	(Proc)
Valid	(Function; virtual)
Validator	(Field)
TFileInputLine	
HandleEvent	Proc; virtual)
Init	(Construc)
TButton	
AmDefault	(Field)
Command	(Field)
Done	(Destruct; virtual)
Draw	(Proc; virtual)
DrawState	(Proc)
Flags	(Field)
GetPalette	(Function; virtual)
HandleEvent	(Proc; virtual)
Init	(Construc)
Load	(Construc)
MakeDefault	(Proc)
Press	(Proc; virtual)
SetState	(Proc; virtual)
Store	(Proc)
Title	(Field)
TCluster	
ButtonState	(Function)
Column	(Function; private)
DataSize	(Function; virtual)
Done	(Destruct; virtual)
DrawBox	(Proc)
DrawMultiBox	(Proc)
EnableMask	(Field)
FindSel	(Function; private)
GetData	(Proc; virtual)
GetHelpCtx	(Function; virtual)
GetPalette	(Function; virtual)
HandleEvent	(Proc; virtual)
Init	(Construc)
Load	(Construc)
Mark	(Function; virtual)
MultiBox	(Proc; virtual)
MultiBox	(Function; virtual)
Press	(Proc; virtual)
Row	(Function; private)
Sel	(Field)